厦门大学计算机科学系研究生课程

获取教材和讲义 PPT 等各种课程资料请访问 http://dblab.xmu.edu.cn/node/422

=课程教材由林子雨老师根据网络资料编著=

=从数据库研究人员的角度阐述大数据技术=

《大数据技术基础》

（版本号：2013 年 12 月 13 日第一版）

厦门大学计算机科学系教师 林子雨 编著

http://www.cs.xmu.edu.cn/linziyu

2013 年 9 月

前言

本教程由厦门大学计算机科学系教师林子雨编著，可以作为计算机专业研究生课程《大数据技术基础》的辅助教材。

本教程共13章，内容包括：第1章 大数据概述、第2章 大数据关键技术与挑战、第3章 Hadoop、第4章 MapReduce、第5章 HDFS、第6章 Zookeeper、第7章 HBase、第8章 流计算、第9章 图计算、第10章 NoSQL数据库、第11章 云数据库、第12章 Google Spanner和第13章 Google Dremel。

本教程由林子雨老师团队合力完成，编写工作分工如下：林子雨负责编写第1章、第2章、第3章、第4章、第5章、第6章、第7章、第10章、第11章和第12章；蔡珉星负责编写第8章，李雨倩负责编写第9章，叶林宝负责编写第13章。

本教程是林子雨通过大量阅读、收集、整理各种资料后精心制作的学习材料，与广大数据库爱好者共享，仅做学术交流之用，请读者不要将此教程用于商业用途。教程中的内容大部分来自网络资料和书籍，一部分是自己撰写。对于自写内容，林子雨老师拥有著作权。感谢林子雨老师团队的多位同学的大量协助工作，包括厦门大学计算机科学系2011级研究生赖明星同学、2012级研究生刘颖杰和叶林宝同学、2013级研究生蔡珉星、李雨倩同学，他们为本教程的撰写做了大量积极的贡献，包括资料收集、整理、讲义PPT制作等。

本教程PDF文档及其全套教学PPT可以通过网络免费下载和使用（下载地址：http://dblab.xmu.edu.cn/node/422）。教程中可能存在一些问题，欢迎读者提出宝贵意见和建议！

本教程已经应用于厦门大学计算机科学系研究生课程《大数据技术基础》，欢迎访问2013班级网站 http://dblab.xmu.edu.cn/node/423。

林子雨的E-mail是：ziyulin@xmu.edu.cn。

林子雨的个人主页是：http://www.cs.xmu.edu.cn/linziyu。

林子雨于厦门大学海韵园

2013年9月

目 录

第 1 章 大数据概述

大数据时代已经到来。最早提出“大数据”时代到来的是全球知名咨询公司麦肯锡，麦肯锡称：“数据，已经渗透到当今每一个行业和业务职能领域，成为重要的生产因素。人们对于海量数据的挖掘和运用，预示着新一波生产率增长和消费者盈余浪潮的到来。”“大数据”在物理学、生物学、环境生态学等领域以及军事、金融、通讯等行业存在已有时日，却因为近年来互联网和信息行业的发展而引起人们关注。

本章内容旨在让大家更好地认识和了解大数据，要点如下：

- 大数据概念
- 大数据的产生和应用
- 大数据作用
- 大数据与大规模数据、海量数据的差别
- 典型的大数据应用实例
- 从数据库到大数据
- 大数据与云计算
- 大数据与物联网
- 对大数据的错误认识
- 大数据技术
- 大数据存储和管理技术
- 大数据生态系统

1.1 大数据概念

随着以博客、社交网络、基于位置的服务 LBS 为代表的新型信息发布方式的不断涌现，以及云计算、物联网等技术的兴起，数据正以前所未有的速度在不断的增长和累积，大数据时代已经来到。

根据 IDC 作出的估测，数据一直都在以每年 50%的速度增长，也就是说每两年就增长一倍（大数据摩尔定律）。这意味着人类在最近两年产生的数据量相当于之前产生的全部数据量，预计到 2020 年，全球将总共拥有 35ZB 的数据量，相较于 2010 年，数据量将增长近 30

倍。这不是简单的数据增多的问题，而是全新的问题。举例来说，在当今全球范围内的工业设备、汽车、电子仪表和装运箱中，都有着无数的数字传感器，这些传感器能测量和交流位置、运动、震动、温度和湿度等数据，甚至还能测量空气中的化学变化。将这些交流传感器与计算智能连接起来，就是目前“物联网”(Internet of Things)或“工业互联网”(Industrial Internet)。在信息获取的问题上取得进步是促进“大数据”趋势发展的重要原因。物联网、云计算、移动互联网、车联网、手机、平板电脑、PC 以及遍布地球各个角落的各种各样的传感器，无一不是数据来源或者承载的方式。

学术界、工业界甚至于政府机构都已经开始密切关注大数据问题，并对其产生浓厚的兴趣。就学术界而言，Nature 早在 2008 年就推出了 Big Data 专刊。计算社区联盟(Computing Community Consortium)在 2008 年发表了报告《Big-Data Computing: Creating revolutionary breakthroughs in commerce, science, and society》，阐述了在数据驱动的研究背景下，解决大数据问题所需的技术以及面临的一些挑战。Science 在 2011 年 2 月推出专刊《Dealing with Data》，主要围绕着科学研究中大数据的问题展开讨论，说明大数据对于科学研究的重要性。美国一些知名的数据管理领域的专家学者则从专业的研究角度出发，联合发布了一份白皮书《Challenges and Opportunities with Big Data》。该白皮书从学术的角度出发，介绍了大数据的产生，分析了大数据的处理流程，并提出大数据所面临的若干挑战。

全球知名的咨询公司麦肯锡(McKinsey)去年 6 月份发布了一份关于大数据的详尽报告《Big data: The next frontier for innovation, competition, and productivity》，对大数据的影响、关键技术和应用领域等都进行了详尽的分析。进入 2012 年以来，大数据的关注度与日俱增。1 月份的达沃斯世界经济论坛上，大数据是主题之一，该次会议还特别针对大数据发布了报告《Big Data, Big Impact: New Possibilities for International Development》，探讨了新的数据产生方式下，如何更好的利用数据来产生良好的社会效益。该报告重点关注了个人产生的移动数据与其他数据的融合与利用。3 月份美国奥巴马政府发布了《大数据研究和发展倡议》(Big Data Research and Development Initiative)，投资 2 亿以上美元，正式启动“大数据发展计划”。计划在科学研究、环境、生物医学等领域利用大数据技术进行突破。奥巴马政府的这一计划被视为美国政府继信息高速公路(Information Highway)计划之后在信息科学领域的又一重大举措。与此同时，联合国一个名为 Global Pulse 的倡议项目在今年 5 月发布报告《Big Data for Development：Challenges & Opportunities》，该报告主要阐述大数据时代各国特别是发展中国家在面临数据洪流(Data Deluge)的情况下所遇到的机遇与挑战，同时还对大数据的应用进行了初步的解读。《纽约时报》的文章《The Age of Big Data》则通过主流媒体的宣传使普通

民众开始意识到大数据的存在，以及大数据对于人们日常生活的影响。

可以看出，“大数据”是时下最火热的 IT 行业词汇。其实，早在 1980 年，著名未来学家阿尔文·托夫勒便在《第三次浪潮》一书中，将大数据热情地赞颂为“第三次浪潮的华彩乐章”。不过，大约从 2009 年开始，“大数据”才成为互联网信息技术行业的流行词汇。

那么，如何给大数据下一个定义呢？一般而言，大家比较认可关于大数据的 4V 说法。大数据的 4 个“V”，或者说是大数据的四个特点，包含四个层面：第一，数据体量巨大。从 TB 级别，跃升到 PB 级别；第二，数据类型繁多。前文提到的网络日志、视频、图片、地理位置信息等等。第三，价值密度低，商业价值高。以视频为例，连续不间断监控过程中，可能有用的数据仅仅有一两秒。第四，处理速度快。1 秒定律。最后这一点也是和传统的数据挖掘技术有着本质的不同。业界将其归纳为 4 个“V”——Volume，Variety，Value，Velocity。

舍恩伯格的《大数据时代》受到了广泛的赞誉，他本人也因此书被视为大数据领域中的领军人物。在舍恩伯格看来，大数据一共具有三个特征：（1）全样而非抽样；（2）效率而非精确；（3）相关而非因果。

第一个特征非常好理解。在过去，由于缺乏获取全体样本的手段，人们发明了“随机调研数据”的方法。理论上，抽取样本越随机，就越能代表整体样本。但问题是获取一个随机样本代价极高，而且很费时。人口调查就是典型一例，一个稍大一点的国家甚至做不到每年都发布一次人口调查，因为随机调研实在是太耗时耗力了。

但有了云计算和数据库以后，获取足够大的样本数据乃至全体数据，就变得非常容易了。谷歌可以提供谷歌流感趋势的原因就在于它几乎覆盖了 7 成以上的北美搜索市场，而在这些数据中，已经完全没有必要去抽样调查这些数据：数据仓库，所有的记录都在那里躺着等待人们的挖掘和分析。

第二点其实建立在第一点的基础上。过去使用抽样的方法，就需要在具体运算上非常精确，因为所谓“差之毫厘便失之千里”。设想一下，在一个总样本为 1 亿人口随机抽取 1000 人，如果在 1000 人上的运算出现错误的话，那么放大到 1 亿中会有多大的偏差。但全样本时，有多少偏差就是多少偏差而不会被放大。谷歌人工智能专家诺维格，在他的论文中写道：大数据基础上的简单算法比小数据基础上的复杂算法更加有效。

数据分析的目的并非仅仅就是数据分析，而是有其它用途，故而时效性也非常重要。精确的计算是以时间消耗为代价的，但在小数据时代，追求精确是为了避免放大的偏差而不得已为之。但在样本=总体的大数据时代，“快速获得一个大概的轮廓和发展脉络，就要比严格的精确性要重要得多”。

第三个特征则非常有趣。相关性表明变量 A 和变量 B 有关，或者说 A 变量的变化和 B 变量的变化之间存在一定的正比（或反比）关系。但相关性并不一定是因果关系（A 未必是 B 的因）。

亚马逊的推荐算法非常有名，它能够根据消费记录来告诉用户你可能会喜欢什么，这些消费记录有可能是别人的，也有可能是该用户历史上的。但它不能说出你为什么会喜欢的原因。难道大家都喜欢购买 A 和 B，就一定等于你买了 A 之后的果就是买 B 吗？未必，但的确需要承认，相关性很高——或者说，概率很大。

舍恩伯格认为，大数据时代只需要知道是什么，而无需知道为什么，就像亚马逊推荐算法一样，知道喜欢 A 的人很可能喜欢 B 但却不知道其中的原因。

1.2 大数据的产生和应用

人类历史上从未有哪个时代和今天一样产生如此海量的数据。数据的产生已经完全不受时间、地点的限制。从开始采用数据库作为数据管理的主要方式开始，人类社会的数据产生方式大致经历了 3 个阶段，而正是数据产生方式的巨大变化才最终导致大数据的产生。

1、运营式系统阶段。数据库的出现使得数据管理的复杂度大大降低，实际中数据库大都为运营系统所采用，作为运营系统的数据管理子系统。比如超市的销售记录系统，银行的交易记录系统、医院病人的医疗记录等。人类社会数据量第一次大的飞跃正是建立在运营式系统开始广泛使用数据库开始。这个阶段最主要特点是数据往往伴随着一定的运营活动而产生并记录在数据库中的，比如超市每销售出一件产品就会在数据库中产生相应的一条销售记录。这种数据的产生方式是被动的。

2、用户原创内容阶段。互联网的诞生促使人类社会数据量出现第二次大的飞跃。但是真正的数据爆发产生于 Web 2.0 时代，而 Web 2.0 的最重要标志就是用户原创内容（UGC,User Generated Content）。这类数据近几年一直呈现爆炸性的增长，主要有两个方面的原因。

首先是以博客、微博为代表的新型社交网络的出现和快速发展，使得用户产生数据的意愿更加强烈。其次就是以智能手机、平板电脑为代表的新型移动设备的出现，这些易携带、全天候接入网络的移动设备使得人们在网上发表自己意见的途径更为便捷。这个阶段数据的产生方式是主动的。

3、感知式系统阶段。人类社会数据量第三次大的飞跃最终导致了大数据的产生，今天

我们正处于这个阶段。这次飞跃的根本原因在于感知式系统的广泛使用。随着技术的发展，人们已经有能力制造极其微小的带有处理功能的传感器，并开始将这些设备广泛的布置于社会的各个角落，通过这些设备来对整个社会的运转进行监控。这些设备会源源不断的产生新数据，这种数据的产生方式是自动的。

简单来说，数据产生经历了被动、主动和自动三个阶段。这些被动、主动和自动的数据共同构成了大数据的数据来源，但其中自动式的数据才是大数据产生的最根本原因。

表 1-1 若干具有代表性的大数据应用及其特征

Applications	Examples	Number of Users	Response Time	Data Scale	Reliability	Accuracy
Scientific Computing	Bioinformatics	Small	Slow	TB	Moderate	Very High
Finance	High-frequency trading	Large	Very Fast	GB	Very High	Very High
Social network	Facebook	Very Large	Fast	PB	High	High
Mobile Data	Mobile phone	Very Large	Fast	TB	High	High
Internet of Things	Sensor network	Large	Fast	TB	High	High
Web Data	News website	Very Large	Fast	PB	High	High
Multimedia	Video site	Very Large	Fast	PB	High	Moderate

正如 Google 的首席经济学家 Hal Varian 所说，数据是广泛可用的，所缺乏的是从中提取出知识的能力。数据收集的根本目的是根据需求从数据中提取有用的知识，并将其应用到具体的领域之中。不同领域的大数据应用有不同的特点，表 1-1 列举了若干具有代表性的大数据应用及其特征。

正是由于大数据的广泛存在，才使得大数据问题的解决很具挑战性。而它的广泛应用，则促使越来越多的人开始关注和研究大数据问题。

1.3 大数据作用

大数据时代已经到来，认同这一判断的人越来越多。那么大数据意味着什么，他到底会改变什么？仅仅从技术角度回答，已不足以解惑。大数据只是宾语，离开了人这个主语，它再大也没有意义。我们需要把大数据放在人的背景中加以透视，理解它作为时代变革力量的所以然。

（1） 变革价值的力量

未来十年，决定中国是不是有大智慧的核心意义标准（那个"思想者"），就是国民幸福。一体现在民生上，二体现在生态上，通过大数据让有意义的事变得明晰，看我们在人与人关

系上，做得是否比以前更有意义。总之，让我们从前 10 年的意义混沌时代，进入未来 10 年意义明晰时代。

（2） 变革经济的力量

生产者是有价值的，消费者是价值的意义所在。有意义的才有价值，消费者不认同的，就卖不出去，就实现不了价值；只有消费者认同的，才卖得出去，才实现得了价值。大数据帮助我们从消费者这个源头识别意义，从而帮助生产者实现价值。这就是启动内需的原理。

（3） 变革组织的力量

随着具有语义网特征的数据基础设施和数据资源发展起来，组织的变革就越来越显得不可避免。大数据将推动网络结构产生无组织的组织力量。最先反映这种结构特点的，是各种各样去中心化的 WEB2.0 应用，如 RSS、维基、博客等。 大数据之所以成为时代变革力量，在于它通过追随意义而获得智慧。

1.4 大数据与大规模数据、海量数据的差别

从对象角度看，大数据是大小超出典型数据库软件采集、储存、管理和分析等能力的数据集合。需要注意的是，大数据并非大量数据的简单无意义的堆积，数据量大并不意味着一定具有可观的利用前景。由于最终目标是从大数据中获取更多有价值的“新”信息，所以必然要求这些大量的数据之间存在着或远或近、或直接或间接的关联性，才具有相当的分析挖掘价值。数据间是否具有结构性和关联性，是 “大数据”与“大规模数据”的重要差别。

从技术角度看，大数据技术是从各种各样类型的大数据中，快速获得有价值信息的技术及其集成。“大数据”与“大规模数据”、“海量数据”等类似概念间的最大区别，就在于“大数据”这一概念中包含着对数据对象的处理行为。为了能够完成这一行为，从大数据对象中快速挖掘更多有价值的信息，使大数据“活起来”，就需要综合运用灵活的、多学科的方法，包括数据聚类、数据挖掘、分布式处理等，而这就需要拥有对各类技术、各类软硬件的集成应用能力。可见，大数据技术是使大数据中所蕴含的价值得以发掘和展现的重要工具。

从应用角度看，大数据是对特定的大数据集合、集成应用大数据技术、获得有价值信息的行为。正由于与具体应用紧密联系，甚至是一对一的联系，才使得“应用”成为大数据不可或缺的内涵之一。

需要明确的是，大数据分析处理的最终目标，是从复杂的数据集合中发现新的关联规则，继而进行深度挖掘，得到有效用的新信息。如果数据量不小，但数据结构简单，重复性

高，分析处理需求也仅仅是根据已有规则进行数据分组归类，未与具体业务紧密结合，依靠已有基本数据分析处理技术已足够，则不能算作是完全的“大数据”，只是“大数据”的初级发展阶段。

1.5 典型的大数据应用实例

1.5.1从谷歌流感趋势看大数据的应用价值

谷歌有一个名为“谷歌流感趋势”的工具，它通过跟踪搜索词相关数据来判断全美地区的流感情况（比如患者会搜索流感两个字）。近日，这个工具发出警告，全美的流感已经进入“紧张”级别。它对于健康服务产业和流行病专家来说是非常有用的，因为它的时效性极强，能够很好地帮助到疾病暴发的跟踪和处理。事实也证明，通过海量搜索词的跟踪获得的趋势报告是很有说服力的，仅波士顿地区，就有 700 例流感得到确认，该地区目前已宣布进入公共健康紧急状态。

这个工具工作的原理大致是这样的：设计人员置入了一些关键词（比如温度计、流感症状、肌肉疼痛、胸闷等），只要用户输入这些关键词，系统就会展开跟踪分析，创建地区流感图表和流感地图。谷歌多次把测试结果（蓝线）与美国疾病控制和预防中心的报告（黄线）做比对，从图 1-1 可知，两者结论存在很大相关性。

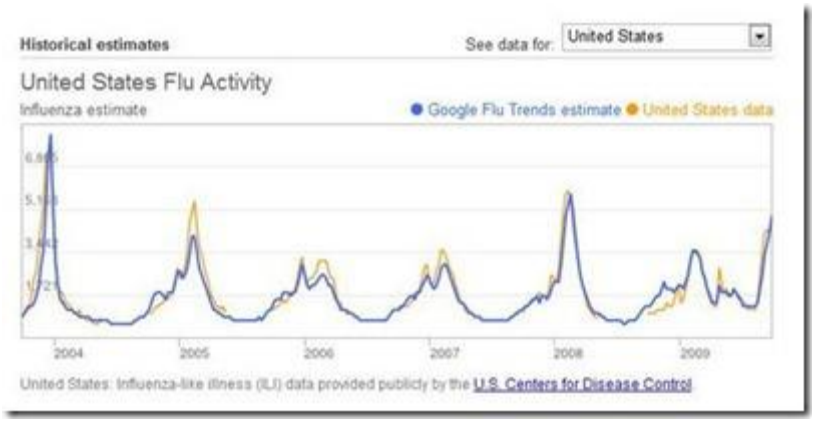

图 1-1 谷歌把测试结果（蓝线）与美国疾病控制和预防中心的报告（黄线）做比对

但它比线下收集的报告强在“时效性”上，因为患者只要一旦自觉有流感症状，在搜索和去医院就诊这两件事上，前者通常是他首先会去做的。就医很麻烦而且价格不菲，如果能自己通过搜索来寻找到一些自我救助的方案，人们就会第一时间使用搜索引擎。故而，还存在一种可能是，医院或官方收集到的病例只能说明一小部分重病患者，轻度患者是不会去医院而成为它们的样本的。

1.5.2 大数据在医疗行业的应用

Seton Healthcare 是采用 IBM 最新沃森技术医疗保健内容分析预测的首个客户。该技术允许企业找到大量病人相关的临床医疗信息，通过大数据处理，更好地分析病人的信息。

在加拿大多伦多的一家医院，针对早产婴儿，每秒钟有超过 3000 次的数据读取。通过这些数据分析，医院能够提前知道哪些早产儿出现问题并且有针对性地采取措施，避免早产婴儿夭折。

大数据让更多的创业者更方便地开发产品，比如通过社交网络来收集数据的健康类 App。也许未来数年后，它们搜集的数据能让医生给你的诊断变得更为精确，比方说不是通用的成人每日三次一次一片，而是检测到你的血液中药剂已经代谢完成会自动提醒你再次服药。

1.5.3 大数据在能源行业的应用

智能电网现在欧洲已经做到了终端，也就是所谓的智能电表。在德国，为了鼓励利用太阳能，会在家庭安装太阳能，除了卖电给你，当你的太阳能有多余电的时候还可以买回来。通过电网收集每隔五分钟或十分钟收集一次数据，收集来的这些数据可以用来预测客户的用电习惯等，从而推断出在未来 2~3 个月时间里，整个电网大概需要多少电。有了这个预测后，就可以向发电或者供电企业购买一定数量的电。因为电有点像期货一样，如果提前买就会比较便宜，买现货就比较贵。通过这个预测后，可以降低采购成本。

维斯塔斯风力系统，依靠的是 BigInsights 软件和 IBM 超级计算机，然后对气象数据进行分析，找出安装风力涡轮机和整个风电场最佳的地点。利用大数据，以往需要数周的分析工作，现在仅需要不足 1 小时便可完成。

1.5.4 大数据在通信行业的应用

XO Communications 通过使用 IBM SPSS 预测分析软件，减少了将近一半的客户流失率。XO 现在可以预测客户的行为，发现行为趋势，并找出存在缺陷的环节，从而帮助公司及时采取措施，保留客户。此外，IBM 新的 Netezza 网络分析加速器，将通过提供单个端到端网络、服务、客户分析视图的可扩展平台，帮助通信企业制定更科学、合理决策。

电信业者透过数以千万计的客户资料，能分析出多种使用者行为和趋势，卖给需要的企

业，这是全新的资料经济。

中国移动通过大数据分析，对企业运营的全业务进行针对性的监控、预警、跟踪。系统在第一时间自动捕捉市场变化，再以最快捷的方式推送给指定负责人，使他在最短时间内获知市场行情。

NTT docomo 把手机位置信息和互联网上的信息结合起来，为顾客提供附近的餐饮店信息，接近末班车时间时，提供末班车信息服务。

1.5.5 大数据在零售业的应用

"我们的某个客户，是一家领先的专业时装零售商，通过当地的百货商店、网络及其邮购目录业务为客户提供服务。公司希望向客户提供差异化服务，如何定位公司的差异化，他们通过从 Twitter 和 Facebook 上收集社交信息，更深入的理解化妆品的营销模式，随后他们认识到必须保留两类有价值的客户：高消费者和高影响者。希望通过接受免费化妆服务，让用户进行口碑宣传，这是交易数据与交互数据的完美结合，为业务挑战提供了解决方案。"Informatica 的技术帮助这家零售商用社交平台上的数据充实了客户主数据，使他的业务服务更具有目标性。

零售企业也监控客户的店内走动情况以及与商品的互动。它们将这些数据与交易记录相结合来展开分析，从而在销售哪些商品、如何摆放货品以及何时调整售价上给出意见，此类方法已经帮助某领先零售企业减少了 17%的存货，同时在保持市场份额的前提下，增加了高利润率自有品牌商品的比例。

1.6 从数据库到大数据

大数据的出现，必将颠覆传统的数据管理方式。在数据来源、数据处理方式和数据思维等方面都会对其带来革命性的变化。本书作者主要从事数据库领域的研究，因此，编写本书时，主要侧重于从数据库存储和管理方面介绍大数据技术。对于数据库研究人员和从业人员而言，必须清楚的是，从数据库(DB)到大数据(BD)，看似只是一个简单的技术演进，但细细考究不难发现两者有着本质上的差别。

如果要用简单的方式来比较传统的数据库和大数据的区别的话，我们认为“池塘捕鱼”和“大海捕鱼”是个很好的类比。“池塘捕鱼”代表着传统数据库时代的数据管理方式，而“大海捕鱼”则对应着大数据时代的数据管理方式，“鱼”是待处理的数据。“捕鱼”环境条

件的变化导致了“捕鱼”方式的根本性差异。这些差异主要体现在如下几个方面：

1、**数据规模**：“池塘”和“大海”最容易发现的区别就是规模。“池塘”规模相对较小，即便是先前认为比较大的“池塘”，譬如 VLDB(Very Large Database)，和“大海”XLDB(Extremely Large Database)相比仍旧偏小。“池塘”的处理对象通常以 MB 为基本单位，而“大海”则常常以 GB，甚至是 TB、PB 为基本处理单位。

2、**数据类型**：过去的“池塘”中，数据的种类单一，往往仅仅有一种或少数几种，这些数据又以结构化数据为主。而在“大海”中，数据的种类繁多，数以千计，而这些数据又包含着结构化、半结构化以及非结构化的数据，并且半结构化和非结构化数据所占份额越来越大。

3、**模式(Schema)和数据的关系**：传统的数据库都是先有模式，然后才会产生数据。这就好比是先选好合适的“池塘”，然后才会向其中投放适合在该“池塘”环境生长的“鱼”。而大数据时代很多情况下难以预先确定模式，模式只有在数据出现之后才能确定，且模式随着数据量的增长处于不断的演变之中。这就好比先有少量的鱼类，随着时间推移，鱼的种类和数量都在不断的增长。鱼的变化会使大海的成分和环境处于不断的变化之中。

4、**处理对象**：在“池塘”中捕鱼，“鱼”仅仅是其捕捞对象。而在“大海”中，“鱼”除了是捕捞对象之外，还可以通过某些“鱼”的存在来判断其他种类的“鱼”是否存在。也就是说传统数据库中数据仅作为处理对象。而在大数据时代，要将数据作为一种资源来辅助解决其他诸多领域的问题。

5、**处理工具**：捕捞“池塘”中的“鱼”，一种渔网或少数几种基本就可以应对，也就是所谓的 One Size Fits All。但是在“大海”中，不可能存在一种渔网能够捕获所有的鱼类，也就是说 No Size Fits All。

从“池塘”到“大海”，不仅仅是规模的变大。传统的数据库代表着数据工程(Data Engineering)的处理方式，大数据时代的数据已不仅仅只是工程处理的对象，需要采取新的数据思维来应对。图灵奖获得者、著名数据库专家 Jim Gray 博士观察并总结人类自古以来，在科学研究上，先后历经了实验、理论和计算三种范式。当数据量不断增长和累积到今天，传统的三种范式在科学研究，特别是一些新的研究领域已经无法很好的发挥作用，需要有一种全新的第四种范式来指导新形势下的科学研究。基于这种考虑，Jim Gray 提出了一种新的数据探索型研究方式，被他自己称之为科学研究的“第四种范式”(The Fourth Paradigm)。

表 1-2 四种范式的比较

Science Paradigms	Time	Methodology
Empirical	Thousand years ago	Describing natural phenomena
Theoretical	Last few hundred years	Using models, generalizations
Computational	Last few decades	Simulating complex phenomena
Data Exploration (eScience)	Today	Data captured by instruments or generated by simulator; Processed by software; Information stored in computer; Scientist analyzes database

四种范式的比较如表 1-2 所示。第四种范式的实质就是从以计算为中心，转变到以数据处理为中心，也就是我们所说的数据思维。这种方式需要我们从根本上转变思维。正如前面提到的“捕鱼”，在大数据时代，数据不再仅仅是“捕捞”的对象，而应当转变成一种基础资源，用数据这种资源来协同解决其他诸多领域的问题。计算社会科学(Computational Social Science)基于特定社会需求，在特定的社会理论指导下，收集、整理和分析数据足迹(data print)，以便进行社会解释、监控、预测与规划的过程和活动。计算社会科学是一种典型的需要采用第四种范式来做指导的科学研究领域。Duncan J. Watts 在《自然》杂志上的文章《A twenty-first century science》也指出借助于社交网络和计算机分析技术，21 世纪的社会科学有可能实现定量化的研究，从而成为一门真正的自然科学。

1.7 大数据与云计算

近几年来，云计算受到学术界和工业界的热捧，随后，大数据横空出世，更是炙手可热。那么，大数据和云计算之间是什么关系呢？

（1）从整体上看，大数据与云计算是相辅相成的

大数据着眼于“数据”，关注实际业务，提供数据采集分析挖掘，看重的是信息积淀，即数据存储能力。云计算着眼于“计算”，关注 IT 解决方案，提供 IT 基础架构，看重的是计算能力，即数据处理能力。没有大数据的信息积淀，则云计算的计算能力再强大，也难以找到用武之地；没有云计算的处理能力，则大数据的信息积淀再丰富，也终究只是镜花水月。

（2）从技术上看，大数据根植于云计算

云计算关键技术中的海量数据存储技术、海量数据管理技术、MapReduce 编程模型，都是大数据技术的基础。

表 1-3 云计算和大数据技术的关系

大数据的关键技术	云计算技术	描述
	虚拟化技术	软硬件隔离，资源整合
	云计算平台管理技术	大规模系统运营，快速故障检测与恢复
	MapReduce编程模型	分布式编程模型，用于并行处理大规模数据集的软件框架
	海量数据存储技术	分布式存储方式存储数据，冗余存储方式保证系统可靠
	海量数据管理技术	NoSQL数据库，进行海量数据管理以便后续分析挖掘

（3）大数据技术与云计算有相同，也有差异

表 1-4 云计算和大数据技术的异同

		大数据	云计算
总体关系		云计算为大数据提供了有力的工具和途径，大数据为云计算提供了很有价值的用武之地	
相同点		1. 都是为数据存储和处理服务 2. 都需要占用大量的存储和计算资源，因而都要用到海量数据存储技术、海量数据管理技术、MapReduce等并行处理技术	
差异点	背景	现有的数据处理技术不能胜任社交网络和物联网产生的大量异构数据，但这些数据存在很大价值	基于互联网的相关服务日益丰富和频繁
	目的	充分挖掘海量数据中的信息	通过互联网更好地调用、扩展和管理计算及存储方面的资源和能力
	对象	数据	IT资源、能力和应用
	推动力量	从事数据存储与处理的软件厂商和拥有大量数据的企业	生产计算及存储设备的厂商、拥有计算及存储资源的企业
	带来的价值	发现数据中的价值	节省IT部署成本

（4）大数据技术与云计算相结合会带来什么？

图 1-2 大数据与云计算的结合

（5）大数据的商业模式与架构----云计算及其分布式结构是重要途径

大数据处理技术正在改变目前计算机的运行模式，正在改变着这个世界。它能处理几乎各种类型的海量数据，无论是微博、文章、电子邮件、文档、音频、视频，还是其它形态的数据。它工作的速度非常快速，实际上几乎实时。它具有普及性，因为它所用的都是最普通低成本的硬件，而云计算将计算任务分布在大量计算机构成的资源池上，使用户能够按需获取计算力、存储空间和信息服务。云计算及其技术给了人们廉价获取巨量计算和存储的能力，云计算分布式架构能够很好地支持大数据存储和处理需求。这样的低成本硬件+低成本软件+低成本运维，更加经济和实用，使得大数据处理和利用成为可能。

1.8 大数据与物联网

物联网是新一代信息技术的重要组成部分，其英文名称是“The Internet of things”。顾名思义，“物联网就是物物相连的互联网”。这有两层意思：第一，物联网的核心和基础仍然是互联网，是在互联网基础上的延伸和扩展的网络；第二，其用户端延伸和扩展到了任何物品与物品之间，进行信息交换和通信。物联网就是“物物相连的互联网”。物联网通过智能感知、识别技术与普适计算、泛在网络的融合应用，被称为继计算机、互联网之后世界信息产业发展的第三次浪潮。物联网是互联网的应用拓展，与其说物联网是网络，不如说物联网

是业务和应用。因此，应用创新是物联网发展的核心，以用户体验为核心的创新 2.0 是物联网发展的灵魂。

物联网架构可分为三层，包括感知层、网络层和应用层：

- 感知层：由各种传感器构成，包括温湿度传感器、二维码标签、RFID 标签和读写器、摄像头、GPS 等感知终端。感知层是物联网识别物体、采集信息的来源；
- 网络层：由各种网络，包括互联网、广电网、网络管理系统和云计算平台等组成，是整个物联网的中枢，负责传递和处理感知层获取的信息；
- 应用层：是物联网和用户的接口，它与行业需求结合，实现物联网的智能应用。

物联网用途广泛，遍及智能交通、环境保护、政府工作、公共安全、平安家居、智能消防、工业监测、环境监测、路灯照明管控、景观照明管控、楼宇照明管控、广场照明管控、老人护理、个人健康、花卉栽培、水系监测、食品溯源、敌情侦查和情报搜集等多个领域。

国际电信联盟于 2005 年的报告曾描绘“物联网”时代的图景：当司机出现操作失误时汽车会自动报警；公文包会提醒主人忘带了什么东西；衣服会“告诉”洗衣机对颜色和水温的要求等等。物联网在物流领域内的应用则比如：一家物流公司应用了物联网系统的货车，当装载超重时，汽车会自动告诉你超载了，并且超载多少，但空间还有剩余，告诉你轻重货怎样搭配；当搬运人员卸货时，一只货物包装可能会大叫“你扔疼我了”，或者说“亲爱的，请你不要太野蛮，可以吗？”；当司机在和别人扯闲话，货车会装作老板的声音怒吼“笨蛋，该发车了！”

物联网把新一代 IT 技术充分运用在各行各业之中，具体地说，就是把感应器嵌入和装备到电网、铁路、桥梁、隧道、公路、建筑、供水系统、大坝、油气管道等各种物体中，然后将“物联网”与现有的互联网整合起来，实现人类社会与物理系统的整合，在这个整合的网络当中，存在能力超级强大的中心计算机集群，能够对整合网络内的人员、机器、设备和基础设施实施实时的管理和控制，在此基础上，人类可以以更加精细和动态的方式管理生产和生活，达到“智慧”状态，提高资源利用率和生产力水平，改善人与自然间的关系。

物联网，移动互联网再加上传统互联网，每天都在产生海量数据，而大数据又通过云计算的形式，将这些数据筛选处理分析，提取出有用的信息，这就是大数据分析。

1.9 对大数据的错误认识

大数据对于悲观者而言，意味着数据存储世界的末日，对乐观者而言，这里孕育了巨

大的市场机会，庞大的数据就是一个信息金矿，随着技术的进步，其财富价值将很快被我们发现，而且越来越容易。

随着物联网和云计算的研究和应用不断深入，对大数据的研究越来越引起广泛的重视，对大数据及其处理技术产生了很多错误的认识，业界有大量关于何谓大数据及它可以做什么的说法，其中有很多是相互矛盾的，都存在一定的片面性，根据 IDC2011 年市场研究报告，主要有三个典型的错误说法：

1） 关系型数据库不能扩展到非常大的数据量，因此不被认为是大数据的技术；

2） 无论工作负载有多大，也无论使用场景如何，Hadoop（或推而广之，任何 Mapreduce 的环境）都是大数据的最佳选择；

3） 基于数据模型的数据库管理系统的时代已经结束了，数据模型必须大数据的方式来建立。

正确的结论是，新型关系型数据库既可解决结构化和非结构化数据，也可满足大数据的数量和速度要求，相比较而言，Hadoop 型解决方案是片面的，不能解决很多的关系型应用环境问题，不一定是最佳选择，大数据管理和处理有更优的解决方案和技术路线。

1.10 大数据技术

大数据本身是一个现象而不是一种技术，伴随着大数据的采集、传输、处理和应用的相关技术就是大数据处理技术，是一系列使用非传统的工具来对大量的结构化、半结构化和非结构化数据进行处理，从而获得分析和预测结果的一系列数据处理技术，或简称大数据技术。

大数据技术主要包括（如图 1-3 所示）：

- 数据采集：ETL 工具负责将分布的、异构数据源中的数据如关系数据、平面数据文件等抽取到临时中间层后进行清洗、转换、集成，最后加载到数据仓库或数据集市中，成为联机分析处理、数据挖掘的基础。
- 数据存取：关系数据库、NoSQL、SQL 等。
- 基础架构：云存储、分布式文件存储等。
- 数据处理：自然语言处理(NLP，Natural Language Processing)是研究人与计算机交互的语言问题的一门学科。处理自然语言的关键是要让计算机"理解"自然语言，所以自然语言处理又叫做自然语言理解(NLU，NaturalLanguage Understanding)，也称为

计算语言学(Computational Linguistics。一方面它是语言信息处理的一个分支，另一方面它是人工智能(AI， Artificial Intelligence)的核心课题之一。

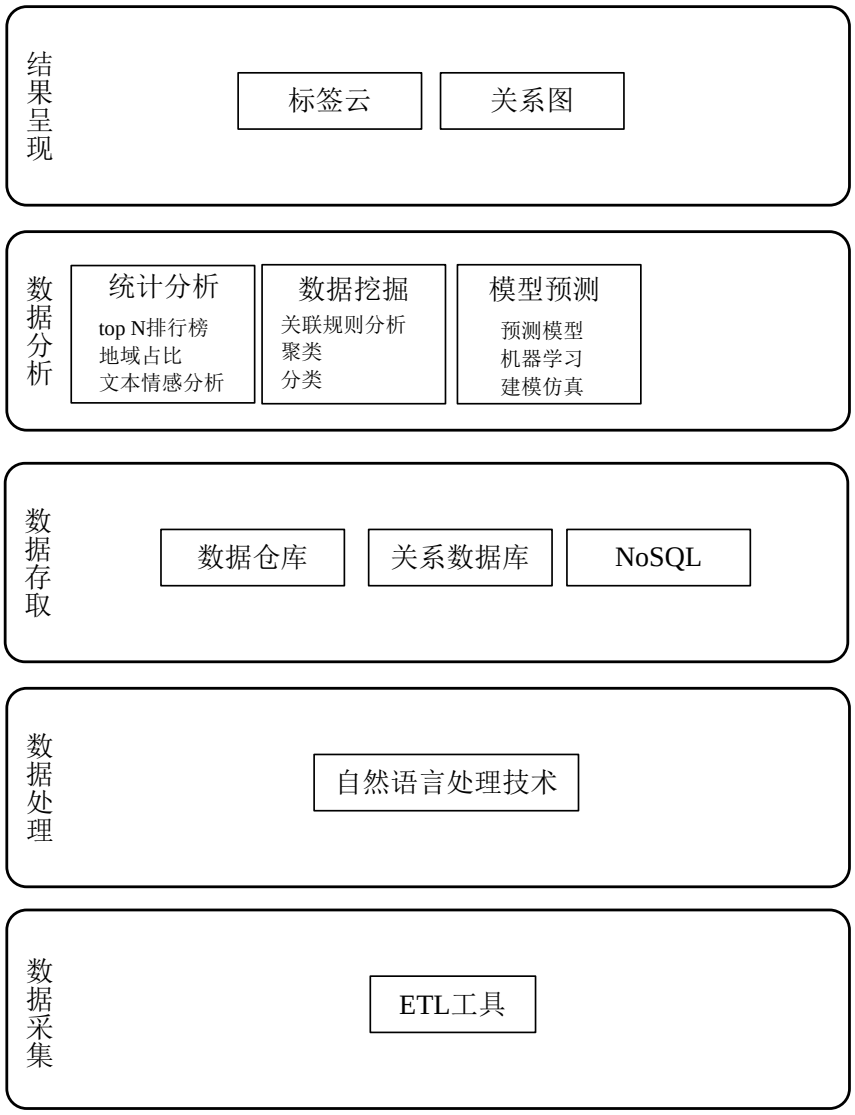

图 1-3 大数据技术内容框架图

- 统计分析：假设检验、显著性检验、差异分析、相关分析、T 检验、方差分析、卡方分析、偏相关分析、距离分析、回归分析、简单回归分析、多元回归分析、逐步回归、回归预测与残差分析、岭回归、logistic 回归分析、曲线估计、因子分析、聚类分析、主成分分析、因子分析、快速聚类法与聚类法、判别分析、对应分析、多元对应分析（最优尺度分析）、bootstrap 技术等等。
- 数据挖掘：分类 （Classification）、估计（Estimation）、预测（Prediction）、相关性分组或关联规则（Affinity grouping or association rules）、聚类（Clustering）、描述和

可视化、Description and Visualization）、复杂数据类型挖掘(Text， Web ，图形图像，视频，音频等)。

- 模型预测：预测模型、机器学习、建模仿真。
- 结果呈现：云计算、标签云、关系图等。

1.11 大数据存储和管理技术

Big Data（大数据技术）是近来的一个技术热点，但从名字就能判断它并不是什么新词。毕竟，大是一个相对概念。历史上，数据库、数据仓库、数据集市等信息管理领域的技术，很大程度上也是为了解决大规模数据的问题。被誉为数据仓库之父的 Bill Inmon 早在 20 世纪 90 年代就经常将 Big Data 挂在嘴边了。

然而，Big Data 作为一个专有名词成为热点，主要应归功于近年来互联网、云计算、移动和物联网的迅猛发展。无所不在的移动设备、RFID、无线传感器每分每秒都在产生数据，数以亿计用户的互联网服务时时刻刻在产生巨量的交互……要处理的数据量实在是太大、增长太快了，而业务需求和竞争压力对数据处理的实时性、有效性又提出了更高要求，传统的常规技术手段根本无法应付。

在这种情况下，技术人员纷纷研发和采用了一批新技术，主要包括分布式缓存、基于 MPP 的分布式数据库、分布式文件系统、各种 NoSQL 分布式存储方案等。

1.11.1 分布式缓存

分布式缓存使用 CARP（Caching Array Routing Protocol）技术，可以产生一种高效率无接缝式的缓存，使用上让多台缓存服务器形同一台，并且不会造成数据重复存放的情况。分布式缓存提供的数据内存缓存可以分布于大量单独的物理机器中。换句话说，分布式缓存所管理的机器实际上就是一个集群。它负责维护集群中成员列表的更新，并负责执行各种操作，比如说在集群成员发生故障时执行故障转移，以及在机器重新加入集群时执行故障恢复。

分布式缓存支持一些基本配置：重复(replicated)、分区(partitioned)和分层(tiered)。重复(Replication)用于提高缓存数据的可用性。在这种情况下，数据将重复缓存在分布式系统的多台成员机器上，这样只要有一个成员发生故障，其他成员便可以继续处理该数据的提供。另一方面，分区(Partitioning)是一种用于实现高可伸缩性的技巧。通过将数据分区存放在许多机器上，内存缓存的大小加随着机器的增加而呈线性增长。结合分区和重复这两种机制创

建出的缓存可同时具备大容量和高可伸缩的特性。分层缓存也称作客户机-服务器(client-server)缓存，它是一种拓扑结构，在该结构中缓存功能将集中于一组机器上。缓存客户机通常并不会亲自执行任何缓存操作，而是连接到缓存并检索或更新其中的数据。分层缓存架构可以包含多层结构。

mem-cached 是 danga.com（运营 LiveJournal 的技术团队）开发的一套分布式内存对象缓存系统，用于在动态系统中减少数据库负载，提升性能。许多 Web 应用程序都将数据保存到 RDBMS 中，应用服务器从中读取数据并在浏览器中显示。但随着数据量的增大，访问的集中，就会出现 REBMS 的负担加重，数据库响应恶化，网站显示延迟等重大影响。Memcached 是高性能的分布式内存缓存服务器。一般的使用目的是通过缓存数据库查询结果，减少数据库的访问次数，以提高动态 Web 应用的速度、提高扩展性。

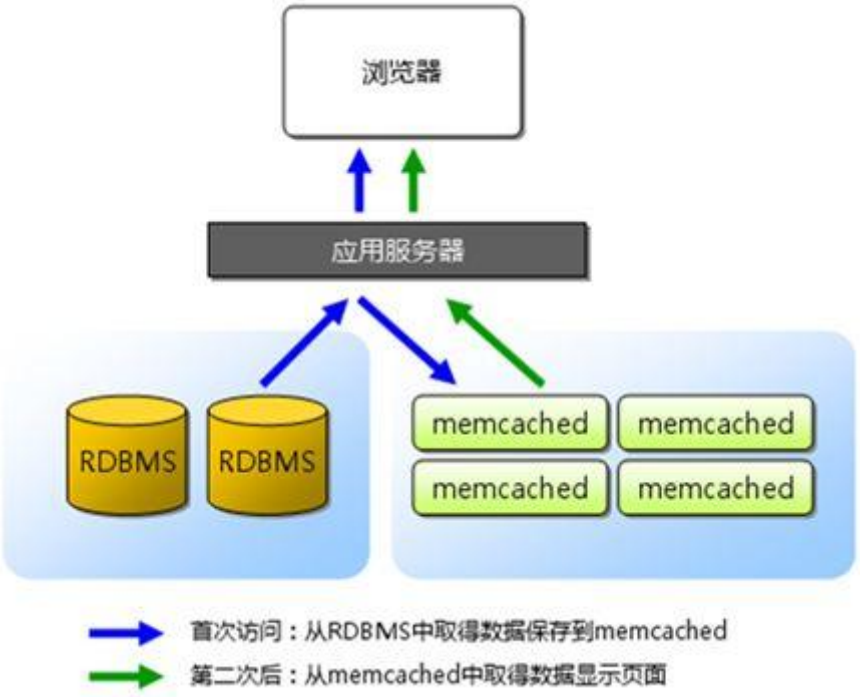

图 1-4 memcached 示意图

Memcached 作为高速运行的分布式缓存服务器具有以下特点：

- 协议简单：memcached 的服务器客户端通信并不使用复杂的 MXL 等格式，而是使用简单的基于文本的协议。
- 基于 libevent 的事件处理：libevent 是个程序库，他将 Linux 的 epoll、BSD 类操作系统的 kqueue 等时间处理功能封装成统一的接口。memcached 使用这个 libevent 库，因此能在 Linux、BSD、Solaris 等操作系统上发挥其高性能。
- 内置内存存储方式：为了提高性能，memcached 中保存的数据都存储在 memcached

内置的内存存储空间中。由于数据仅存在于内存中，因此重启 memcached，重启操作系统会导致全部数据消失。另外，内容容量达到指定的值之后 memcached 回自动删除不适用的缓存。

- Memcached 不互通信的分布式：memcached 尽管是“分布式”缓存服务器，但服务器端并没有分布式功能。各个 memcached 不会互相通信以共享信息。它的分布式主要是通过客户端实现的。

memcached 处理的原子是每一个（Key，Value）对（以下简称 KV 对），Key 会通过一个 hash 算法转化成 hash-Key，便于查找、对比以及做到尽可能的散列。同时，memcached 用的是一个二级散列，通过一张大 hash 表来维护。

memcached 由两个核心组件组成：服务端（ms）和客户端（mc），在一个 memcached 的查询中，ms 先通过计算 Key 的 hash 值来确定 KV 对所处在的 ms 位置。当 ms 确定后，mc 就会发送一个查询请求给对应的 ms，让它来查找确切的数据。因为这之间没有交互以及多播协议，所以 memcached 交互带给网络的影响是最小化的。

MemcacheDB 是一个分布式、Key-Value 形式的持久存储系统。它不是一个缓存组件，而是一个基于对象存取的、可靠的、快速的持久存储引擎。协议与 memcached 一致（不完整），所以，很多 memcached 客户端都可以跟它连接。MemcacheDB 采用 Berkeley DB 作为持久存储组件，因此，很多 Berkeley DB 的特性它都支持。

类似这样的产品也很多，如淘宝 Tair 就是 Key-Value 结构存储，在淘宝得到了广泛使用。后来 Tair 也做了一个持久化版本，思路基本与新浪 MemcacheDB 一致。

1.11.2 分布式数据库

分布式数据库系统通常使用较小的计算机系统，每台计算机可单独放在一个地方，每台计算机中都有 DBMS 的一份完整拷贝副本，并具有自己局部的数据库，位于不同地点的许多计算机通过网络互相连接，共同组成一个完整的、全局的大型数据库。

分布式数据库系统是数据库系统与计算机网络相结合的产物。分布式数据库系统产生于 1970 年代末期，在 1980 年代进入迅速成长阶段。由于数据库应用需求的拓展和计算机硬件环境的改变，计算机网络与数字通信技术的飞速发展，卫星通信、蜂窝通信、计算机局域网、广域网和 Internet 的迅速发展，使得分布式数据库系统应运而生，并成为计算机技术最活跃的研究领域之一。

分布式数据库系统符合信息系统应用的需求，符合当前企业组织的管理思想和管理方式。对于地域上分散而管理上又相对集中的大企业而言，数据通常是分布存储在不同地理位置，每个部门都会负责维护与自己工作相关的数据。整个企业的信息就被分隔成多个“信息孤岛”。分布式数据库为这些信息孤岛提供了一座桥梁。分布式数据库的结构能够反映当今企业组织的信息数据结构，本地数据保存在本地维护，而又可以在需要时存取异地数据。也就是说，既需要有各部门的局部控制和分散管理，同时也需要整个组织的全局控制和高层次的协同管理。这种协同管理要求各部门之间的信息既能灵活交流与共享，又能统一管理和使用，自然而然就提出了对分布式数据库系统的需求。随着应用需求的扩大和要求的提高，人们越来越认识到集中式数据库的局限性，迫切需要把这些子部门的信息通过网络连接起来，组成一个分布式数据库。

世界上第一个分布式数据库系统 SDD-1，是由美国计算机公司于 1976 年-1978 年设计的，并于 1979 年在 DEC-10 和 DEC-20 计算机上面实现。

Spanner 是一个可扩展、多版本、全球分布式并支持同步复制的分布式数据库。它是 Google 的第一个可以全球扩展并且支持外部一致性事务的分布式数据库。Spanner 能做到这些，离不开一个用 GPS 和原子钟实现的时间 API。这个 API 能将数据中心之间的时间同步精确到 10ms 以内。因此，Spanner 有几个给力的功能：无锁读事务、 原子模式修改、读历史数据无阻塞。

1.11.3 分布式文件系统

谈到分布式文件系统，不得不提的是 Google 的 GFS。基于大量安装有 Linux 操作系统的普通 PC 构成的集群系统，整个集群系统由一台 Master（通常有几台备份）和若干台 TrunkServer 构成。GFS 中文件被分成固定大小的 Trunk 分别存储在不同的 TrunkServer 上，每个 Trunk 有多份（通常为 3 份）拷贝，也存储在不同的 TrunkServer 上。Master 负责维护 GFS 中的 Metadata，即文件名及其 Trunk 信息。客户端先从 Master 上得到文件的 Metadata，根据要读取的数据在文件中的位置与相应的 TrunkServer 通信，获取文件数据。

在 Google 的论文发表后，就诞生了 Hadoop。截至今日，Hadoop 被很多中国最大互联网公司所追捧，百度的搜索日志分析，腾讯、淘宝和支付宝的数据仓库都可以看到 Hadoop 的身影。

Hadoop 具备低廉的硬件成本、开源的软件体系、较强的灵活性、允许用户自己修改代

码等特点，同时能支持海量数据存储和计算任务。

1.11.4 NoSQL

NoSQL 数据库，指的是非关系型的数据库。随着互联网 web2.0 网站的兴起，传统的关系数据库在应付 web2.0 网站，特别是超大规模和高并发的 SNS 类型的 web2.0 纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。

现今的计算机体系结构在数据存储方面要求具备庞大的水平扩展性（horizontal scalability，是指能够连接多个软硬件的特性，这样可以将多个服务器从逻辑上看成一个实体），而 NoSQL 致力于改变这一现状。目前 Google 的 BigTable 和 Amazon 的 Dynamo 使用的就是 NoSQL 型数据库。2010 年下半年，Facebook 选择 HBase 来做实时消息存储系统，替换原来开发的 Cassandra 系统。这使得很多人开始关注 NoSQL 型数据库 HBase。Facebook 选择 HBase 是基于短期小批量临时数据和长期增长的很少被访问到的数据这两个需求来考虑的。

HBase 是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用 HBase 技术可在廉价 PC Server 上搭建大规模结构化存储集群。HBase 是 BigTable 的开源实现，使用 HDFS 作为其文件存储系统。Google 运行 MapReduce 来处理 BigTable 中的海量数据，HBase 同样利用 MapReduce 来处理 HBase 中的海量数据；BigTable 利用 Chubby 作为协同服务，HBase 则利用 Zookeeper 作为对应。

1.12 大数据生态系统

以下是福布斯专栏作家 Dave Feinleib 绘制的一张大数据生态系统图谱，非常有参考价值。该信息图中涉及的公司、产品和技术：

- Splunk,Loggly,Sumologic
- Predictive Policing,BloomReach
- Gnip,Datasift,Space Curve,Inrix
- Oracle Hyperion, SAP BusinessObjects，Microsoft Business Intelligence, IBM Cognos, SAS,MicroStrategy, GoodData
- Tableau Software, Palantir, MetaMarkets, Teradata Aster, Visual.ly, KarmaSphere, EMC

Greenplum, Platfora, ClearStory

- HortonWorks, Cloudera, MapR, Vertica
- Couchbase, Teradata, 10gen, Hadapt
- Amazon Web Services Elastic MapReduce, Infochimps, Microsoft Windows Azure
- Oracle, Microsoft SQL Server, MySQL, PostgreSQL, memsql
- Hadoop, MapReduce, Hbase, Cassandra

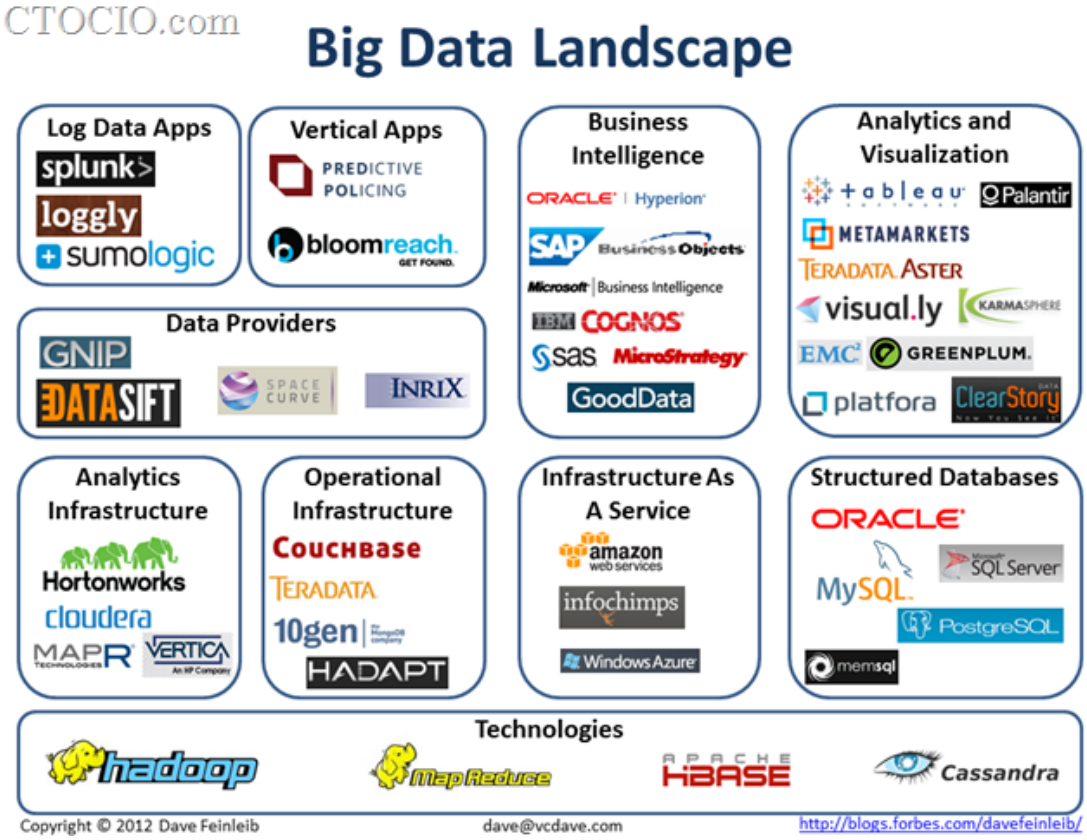

图 1-5 大数据生态系统

本章小结

本章介绍了大数据的概念、产生、应用与作用，并介绍了大数据与大规模数据、海量数据的差别，同时给出了大型的大数据应用案例；同时，分析了大数据与云计算、物联网的相互关系，并提到了一些关于大数据的错误认识；介绍了大数据技术以及大数据存储与管理技术；最后描述了大数据生态系统。

参考文献

[1] 孟小峰，慈祥. 大数据管理：概念、技术与挑战. 计算机学报, 2013 年第 8 期.

[2]关志刚. 信息图：大数据企业生态图谱. IT 经理网. http://www.ctocio.com/bigdata/7028.html

[3] 百度百科. 物联网.

[4] 邵佩英. 分布式数据库系统及其应用，科学出版社.

[5] 其他网络来源.

第 2 章 大数据关键技术与挑战

大数据价值的完整体现需要多种技术的协同。文件系统提供最底层存储能力的支持。为了便于数据管理，需要在文件系统之上建立数据库系统。通过索引等的构建，对外提供高效的数据查询等常用功能。最终通过数据分析技术从数据库中的大数据提取出有益的知识。

本章内容首先介绍大数据处理基本流程，然后介绍大数据处理模型，接下来阐述了大数据关键技术，并讨论了大数据时代面临的新挑战，内容要点如下：

- 大数据处理的基本流程
- 大数据处理模型
- 大数据关键技术
- 大数据处理工具
- 大数据时代面临的新挑战

2.1 大数据处理的基本流程

大数据的数据来源广泛，应用需求和数据类型都不尽相同，但是最基本的处理流程一致。海量 Web 数据的处理是一类非常典型的大数据应用，从中可以归纳出大数据处理的最基本流程，如图 2-1 所示。

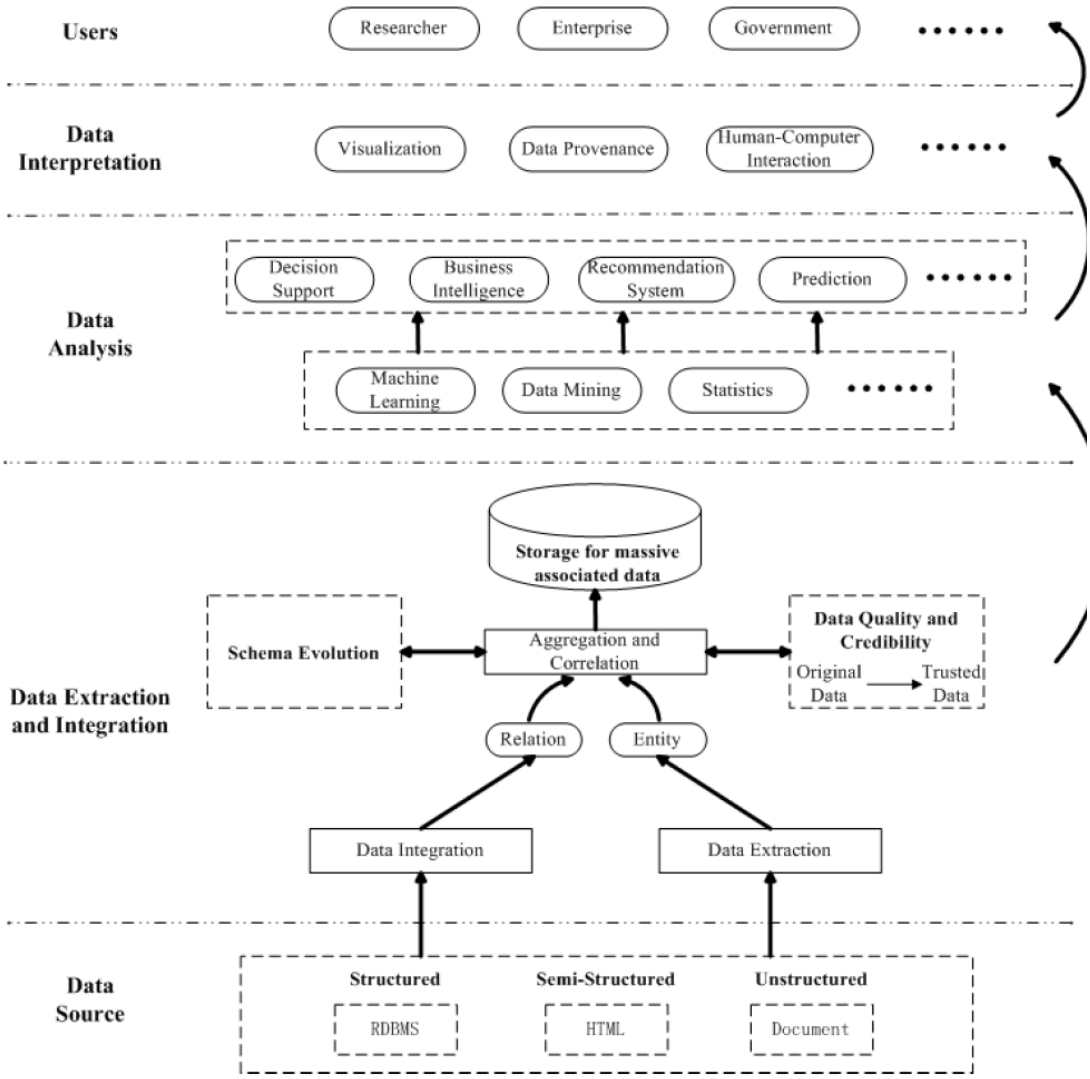

图 2-1 大数据处理的基本流程

整个大数据的处理流程可以定义为：在合适工具的辅助下，对广泛异构的数据源进行抽取和集成，结果按照一定的标准进行统一存储，并利用合适的数据分析技术对存储的数据进行分析，从中提取有益的知识并利用恰当的方式将结果展现给终端用户。具体来说，可以分为数据抽取与集成、数据分析以及数据解释。

2.1.1 数据抽取与集成

大数据的一个重要特点就是多样性，这就意味着数据来源极其广泛，数据类型极为繁杂。这种复杂的数据环境给大数据的处理带来极大的挑战。要想处理大数据，首先必须对所需数据源的数据进行抽取和集成，从中提取出关系和实体，经过关联和聚合之后采用统一定义的结构来存储这些数据。在数据集成和提取时需要对数据进行清洗，保证数据质量及可信

性。

同时还要特别注意大数据时代模式和数据的关系，大数据时代的数据往往是先有数据再有模式，且模式是在不断的动态演化之中的。数据抽取和集成技术不是一项全新的技术，传统数据库领域已对此问题有了比较成熟的研究。随着新的数据源的涌现，数据集成方法也在不断的发展之中。从数据集成模型来看，现有的数据抽取与集成方式可以大致分为以下四种类型：基于物化或是 ETL 方法的引擎(Materialization or ETL engine)、基于联邦数据库或中间件方法的引擎(Federation engine or Mediator)、基于数据流方法的引擎(Stream engine)及基于搜索引擎的方法(Search engine)。

[拓展阅读]数据集成方式

数据集成方式包括数据整合、数据联邦、数据传播和混合方法等四种：

（1）数据整合（Data Consolidation）：不同数据源的数据被物理地集成到数据目标。利用 ETL 工具把数据源中的数据批量地加载到数据仓库，就属于数据整合的方式。

（2）数据联邦（Data Federation）：在多个数据源的基础上建立一个统一的逻辑视图，对外界应用屏蔽数据在各个数据源的分布细节。对于这些应用而言，只有一个统一的数据访问入口，但是实际上，被请求的数据只是逻辑意义上的集中，在物理上仍然分布在各个数据源中，只有被请求时，才临时从不同数据源获取相关数据，进行集成后提交给数据请求者。当数据整合方式代价太大或者为了满足一些突发的实时数据需求时，可以考虑采用数据联邦的方式建立企业范围内的全局统一数据视图。

（3）数据传播（Data Propagation）：数据在多个应用之间的传播。比如，在企业应用集成（EAI）解决方案中，不同应用之间可以通过传播消息进行交互。

（4）混合方式（A Hybrid Approach）：在这种方式中，对于那些不同应用都使用的数据采用数据整合的方式进行集成，而对那些只有特定应用才使用的数据则采用数据联邦的方式进行集成。

2.1.2 数据分析

数据分析是整个大数据处理流程的核心，因为大数据的价值产生于分析过程。从异构数据源抽取和集成的数据构成了数据分析的原始数据。根据不同应用的需求可以从这些数据中选择全部或部分进行分析。传统的分析技术如数据挖掘、机器学习、统计分析等在大数据时

代需要做出调整，因为这些技术在大数据时代面临着一些新的挑战，主要有：

- **数据量大并不一定意味着数据价值的增加，相反这往往意味着数据噪音的增多**。因此在数据分析之前必须进行数据清洗等预处理工作，但是预处理如此大量的数据对于机器硬件以及算法都是严峻的考验。
- **大数据时代的算法需要进行调整**。首先，大数据的应用常常具有实时性的特点，算法的准确率不再是大数据应用的最主要指标。很多场景中算法需要在处理的实时性和准确率之间取得一个平衡，比如在线的机器学习算法(online machine learning)。其次，云计算是进行大数据处理的有力工具，这就要求很多算法必须做出调整以适应云计算的框架，算法需要变得具有可扩展性。最后，在选择算法处理大数据时必须谨慎，当数据量增长到一定规模以后，可以从小量数据中挖掘出有效信息的算法并不一定适用于大数据。统计学中的邦弗朗尼原理(Bonferroni's Principle)就是一个典型的例子。
- **数据结果好坏的衡量**。得到分析结果并不难，但是结果好坏的衡量却是大数据时代数据分析的新挑战。大数据时代的数据量大、类型庞杂，进行分析的时候往往对整个数据的分布特点掌握得不太清楚，这会导致最后在设计衡量的方法以及指标的时候遇到诸多困难。大数据分析已被广泛应用于诸多领域，典型的有推荐系统、商业智能、决策支持等。

[拓展阅读] 邦弗朗尼原理

假定人们有一定量的数据，并期望从该数据中找到某个特定类型的事件。即使数据完全随机，也可以期望该类型事件会发生。随着数据规模的增长，这类事件出现的数目也随之上升。任何随机数据往往都会有一些不同寻常的特征，这些特征看上去虽然很重要，但是实际上并不重要，除此之外，别无他由，从这个意义上说，这些事件的出现纯属"臆造"。统计学上有一个称为邦弗朗尼校正（Bonferroni correction）的定理，该定理给出一个在统计上可行的方法来避免在搜索数据时出现的大部分"臆造"正响应。这里并不打算介绍定理的统计细节，只给出一个非正式的称为邦弗朗尼原理的版本，该原理可以帮助我们避免将随机出现看成真正出现。在数据随机性假设的基础上，可以计算所寻找事件出现次数的期望值。如果该结果显著高于你所希望找到的真正实例的数目，那么可以预期，寻找到的几乎任何事物都是臆造的，也就是说，它们是在统计上出现的假象，而不是你所寻找事件的凭证。上述观察现象是邦弗朗尼原理的非正式

阐述。简单地说，你假设：特定事件的发生预示着特定内容。如果特定事件(例如：在酒店中聚会)发生的概率乘以样本空间得到的数目远远大与你期望的特定内容(例如：歹徒)的数目，那么你的假设是错的。总之，我们不能指望通过大规模统计来发现一些很“稀有”的事情或者规律。例如：恐怖袭击这样的事情，多少年都遇不到一次。通过对某些数据的大规模统计，可能推断出每年要发生很多起恐怖袭击，这本身就不现实。

2.1.3 数据解释

数据分析是大数据处理的核心，但是用户往往更关心结果的展示。如果分析的结果正确但是没有采用适当的解释方法，则所得到的结果很可能让用户难以理解，极端情况下甚至会误导用户。数据解释的方法很多，比较传统的就是以文本形式输出结果或者直接在电脑终端上显示结果。这种方法在面对小数据量时是一种很好的选择。但是大数据时代的数据分析结果往往也是海量的，同时结果之间的关联关系极其复杂，采用传统的解释方法基本不可行。可以考虑从下面两个方面提升数据解释能力：

- **引入可视化技术**。可视化作为解释大量数据最有效的手段之一率先被科学与工程计算领域采用。通过对分析结果的可视化用形象的方式向用户展示结果，而且图形化的方式比文字更易理解和接受。常见的可视化技术有标签云(Tag Cloud)、历史流(history flow)、空间信息流(Spatial information flow)等。可以根据具体的应用需要选择合适的可视化技术。
- **让用户能够在一定程度上了解和参与具体的分析过程**。这个既可以采用人机交互技术，利用交互式的数据分析过程来引导用户逐步地进行分析，使得用户在得到结果的同时更好的理解分析结果的由来。也可以采用数据起源技术，通过该技术可以帮助追溯整个数据分析的过程，有助于用户理解结果。

2.2 大数据处理模型

2.2.1 大数据之快和处理模型

天下武功，唯快不破。这句话源自《拳经》，经过金山公司董事长雷军等人的演绎，几乎成了互联网时代商业致胜的不二法则。那么，大数据的快又从何说起呢？

“快”，来自几个朴素的思想：

- **时间就是金钱**。时间在分母上，越小，单位价值就越大。面临同样大的数据矿山，“挖矿”效率是竞争优势。Zara 与 H&M 有相似的大数据供应，Zara 胜出的原因毫无疑问就是“快”。
- **像其它商品一样，数据的价值会折旧**。过去一天的数据，比过去一个月的数据可能都更有价值。更普遍意义上，它就是时间成本的问题：等量数据在不同时间点上价值不等。NewSQL 的先行者 VoltDB 发明了一个概念叫做 Data Continuum：数据存在于一个连续时间轴（time continuum）上，每一个数据项都有它的年龄，不同年龄的数据有不同的价值取向，“年轻”（最近）时关注个体的价值，“年长”（久远）时注重集合价值。
- **数据跟新闻和金融行情一样，具有时效性**。炒股软件免费版给你的数据有十几秒的延迟，这十几秒是快速猎食者宰割散户的机会；而华尔街大量的机构使用高频机器交易（70%的成交量来自高频交易），能发现微秒级交易机会的吃定毫秒级的。物联网这块，很多传感器的数据，产生几秒之后就失去意义了。美国国家海洋和大气管理局的超级计算机能够在日本地震后 9 分钟计算出海啸的可能性，但 9 分钟的延迟对于瞬间被海浪吞噬的生命来说还是太长了。

大家知道，购物篮分析是沃尔玛横行天下的绝技，其中最经典的就是关联产品分析：从大家耳熟能详的“啤酒加尿布”，到飓风来临时的“馅饼（pop-tarts）加手电筒”和“馅饼加啤酒”。可是，此“购物篮”并非顾客拎着找货的那个，而是指你买完帐单上的物品集合。对于快消品等有定期消费规律的产品来说，这种“购物篮”分析尚且有效，但对绝大多数商品来说，找到顾客“触点（touch points）”的最佳时机并非在结帐以后，而是在顾客还领着篮子扫街逛店的正当时。电子商务具备了这个能力，从点击流（clickstream）、浏览历史和行为（如放入购物车）中实时发现顾客的即时购买意图和兴趣。这就是“快”的价值。

设想我们站在某个时间点上，背后是静静躺着的老数据，面前是排山倒海扑面而来的新数据。在令人窒息的数据海啸面前，我们的数据存储系统如同一个小型水库，而数据处理系统则可以看作是水处理系统。数据涌入这个水库， 如果不能很快处理，只能原封不动地排出。对于数据拥有者来说，除了付出了存储设备的成本，没有收获任何价值。

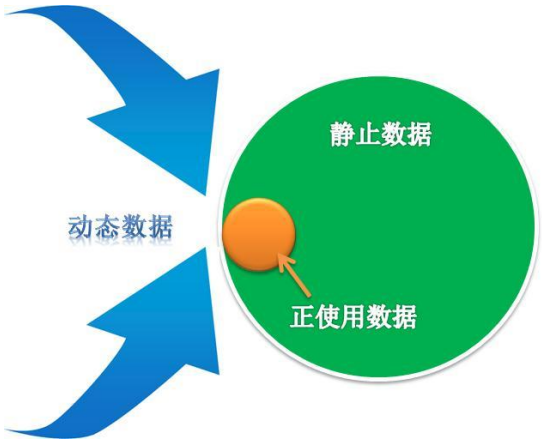

图 2-2 数据的三种状态

如图 2-2 所示，按照数据的三状态定义，水库里一平如镜（非活跃）的水是“静止数据（data at rest）”，水处理系统中上下翻动的水是“正使用数据（data inuse）”，汹涌而来的新水流就是“动态数据（data in motion）”。

“快”说的是两个层面：

- **一个是“动态数据”来得快**。动态数据有不同的产生模式。有的是“爆发(burst)”模式，极端的例子如欧洲核子研究中心（CERN）的大型强子对撞机(Large Hadron Collider，简称 LHC)，此机不撞则已，一撞惊人，工作状态下每秒产生 PB 级的数据。也有的动态数据是涓涓细流的模式，典型的如 clickstream、日志、RFID 数据、GPS 位置信息和 Twitter 的 firehose 流数据等。
- **二是对“正使用数据”处理得快**。水处理系统可以从水库调出水来进行处理（“静止数据”转变为“正使用数据”），也可以直接对涌进来的新水流处理（“动态数据”转变为“正使用数据”）。这对应着两种大相迥异的处理范式：批处理和流处理。

如图 2-3 所示，左半部是批处理：以“静止数据”为出发点，数据是任尔东西南北风、我自岿然不动，处理逻辑进来，算完后价值出去。Hadoop 就是典型的批处理范式：HDFS 存放已经沉淀下来的数据，MapReduce 的作业调度系统把处理逻辑送到每个节点进行计算。这非常合理，因为搬动数据比发送代码更昂贵。

右半部则是流数据处理范式。这次不动的是逻辑，“动态数据”进来，计算完后价值留下，原始数据加入“静止数据”，或索性丢弃。流处理品类繁多，包括传统的消息队列（绝大多数的名字以 MQ 结尾）、事件流处理（Event Stream Processing）/复杂事件处理（Complex Event Processing 或 CEP）（如 Tibco 的 BusinessEvents 和 IBM 的 InfoStreams）、分布式发布/订阅系

统（如 Kafka）、专注于日志处理的（如 Scribe 和 Flume）、通用流处理系统（如 Storm 和 S4）等。

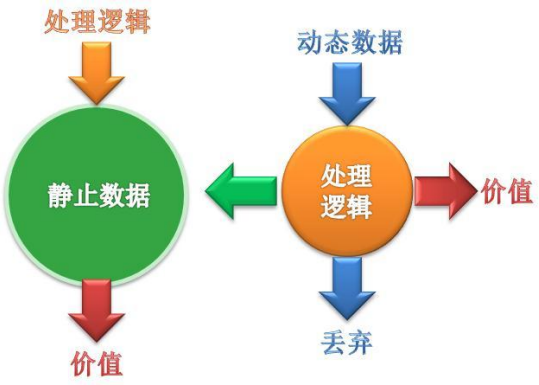

图 2-3 数据的两种处理模型

这两种范式与我们日常生活中的两种信息处理习惯相似：有些人习惯先把信息存下来（如书签、To Do 列表、邮箱里的未读邮件），稍后一次性地处理掉（也有可能越积越多，旧的信息可能永远不会处理了）；有些人喜欢任务来一件做一件，信息来一点处理一点，有的直接过滤掉，有的存起来。

没有定规说哪种范式更好，对于批量数据，多数是先进入存储系统，然后再来处理，因此以批处理范式为主；而对于流数据，多采用流范式。传统上认为流处理的方式更快，但流范式能处理的数据常常局限于最近的一个数据窗口，只能获得实时智能（real-time intelligence），不能实现全时智能（all-time intelligence）。批处理擅长全时智能，但翻江倒海捣腾数据肯定慢，所以亟需把批处理加速。

两种范式常常组合使用，而且形成了一些定式：

- **流处理作为批处理的前端**：比如大型强子对撞机，每秒 PB 级的数据先经过流处理范式进行过滤，只有那些科学家感兴趣的撞击数据保留下来进入存储系统，留待批处理范式处理。这样，欧洲核子研究中心每年的新增存储量可以减到 25PB。
- **流处理与批处理肩并肩**：流处理负责动态数据和实时智能，批处理负责静止数据和历史智能，实时智能和历史智能合并成为全时智能。

那么，如何实现“快”的数据处理呢？

首先，“快”是个相对的概念，可以是实时，也可以秒级、分钟级、小时级、天级甚至更长的延迟。实现不同级别的“快”采用的架构和付出的代价也不一样。所以，对于每一个面

临“快”问题的决策者和架构师来说，第一件事情就是要搞清楚究竟要多“快”。“快”无止境，找到足够“快”的那个点，那就够了。

其次，考虑目前的架构是不是有潜力改造到足够“快”。很多企业传统的关系型数据库中数据量到达 TB 级别，就慢如蜗牛了。在转向新的架构（如 NoSQL 数据库）之前，可以先考虑分库分表（sharding）和内存缓存服务器（如 memcached）等方式延长现有架构的生命。

如果预测未来数据的增长必将超出现有架构的上限，那就要规划新的架构了。这里不可避免要做出选择，或者选择流处理结构，或者选择批处理结构，当然也可以选择两者兼具。Intel 有一位老法师说：any big data platform needs to be architected for particular problems（任何一个大数据平台都需要为特定的问题度身定做）。这是非常有道理的。为什么呢？比如说大方向决定了要用流处理架构，落实到具体产品少说有上百种，所以要选择最适合的流处理产品。再看批处理架构，MapReduce 也不能包打天下，碰到多迭代、交互式计算就无能为力了；NoSQL 更是枝繁叶茂，有名有姓的 NoSQL 数据库好几十种。这时候请一个好的大数据咨询师很重要，让他帮助企业选择一个量身定制的大数据解决方案。

总体上讲，还是有一些通用的技术思路来实现“快”：

- **如果数据流入量太大，在前端就地采用流处理进行即时处理、过滤掉非重要数据**。
- **把数据预处理成适于快速分析的格式**。预处理常常比较耗时，但对不常改动的惰性数据，预处理的代价在长期的使用中可以被分摊到很小，甚至可以忽略不计。谷歌的 Dremel 就是把只读的嵌套数据转成类似于列式数据库的形式，实现了 PB 级数据的秒级查询。
- **增量计算--也即先顾眼前的新数据，再去更新老数据**。对传统的批处理老外叫做 reboil the ocean，每次计算都要翻江倒海把所有数据都捣腾一遍，自然快不了。而增量计算把当前重点放在新数据上，先满足“快”；同时抽空把新数据（或新数据里提炼出来的信息）更新到老数据（或历史信息库）中，又能满足“全”。谷歌的 Web 索引自 2010 年起从老的 MapReduce 批量系统升级成新的增量索引系统，能够极大地缩短网页被爬虫爬到和被搜索到之间的延迟。前面说的“流处理和批处理肩并肩”也是一种增量计算。
- **很多批处理系统慢的根源是磁盘和 I/O，把原始数据和中间数据放在内存里，一定能极大地提升速度**。这就是内存计算（In-memory computing）。内存计算最简单的形式是内存缓存，Facebook 超过 80%的活跃数据就在 memcached 里。比较复杂的

有内存数据库和数据分析平台，如 SAP 的 HANA，NewSQL 的代表 VoltDB 和伯克利的开源内存计算框架 Spark（Intel 也开始参与）。斯坦福的 John Ousterhout（Tcl/Tk 以及集群文件系统 Lustre 的发明者）搞了个更超前的 RAMCloud，号称所有数据只生活在内存里。未来新的非易失性内存（断电数据不会丢失）会是个游戏规则改变者。Facebook 在 3 月宣布了闪存版的 Memcached，叫 McDipper，比起单节点容量可以提升 20 倍，而吞吐量仍能达到每秒数万次操作。另一种非易失性内存，相变内存（Phase Change Memory），在几年内会商用，它的每比特成本可以是 DRAM 的 1/10，性能比 DRAM 仅慢 2-10 倍，比现今的闪存（NAND）快 500 倍，寿命长 100 倍。除内存计算外，还有其它的硬件手段来加速计算、存储和数据通讯，如 FPGA（IBM 的 Netezza 和 Convey 的 Hybrid-Core），SSD 和闪存卡（SAP HANA 和 Fusion IO），压缩 PCIe 卡，更快和可配置的互联（Infiniband 的 RDMA 和 SeaMicro SM15000 的 Freedom Fabrics）等。此处不再细表。

- **降低对精确性的要求**。大体量、精确性和快不可兼得，顶多取其二。如果要在大体量数据上实现“快”，必然要适度地降低精确性。对数据进行采样后再计算就是一种办法，伯克利 BlinkDB 通过独特的采用优化技术实现了比 Hive 快百倍的速度，同时能把误差控制在 2-10%。

2.2.2 流处理

大数据的应用类型很多，主要的处理模式可以分为流处理(Stream Processing)和批处理(Batch Processing)两种。批处理是先存储后处理(Store-then-process)，而流处理则是直接处理(Straight-through processing)。

流处理的基本理念是数据的价值会随着时间的流逝而不断减少。因此尽可能快的对最新的数据做出分析并给出结果是所有流数据处理模式的共同目标。需要采用流数据处理的大数据应用场景主要有网页点击数的实时统计、传感器网络、金融中的高频交易等。

流处理的处理模式将数据视为流，源源不断的数据组成了数据流。当新的数据到来时就立刻处理并返回所需的结果。图 2-4 是流处理中基本的数据流模型：

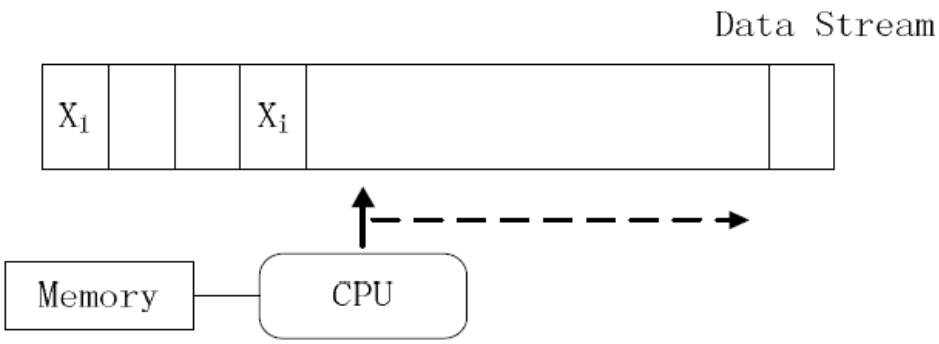

图 2-4 流处理中基本的数据流模型

数据的实时处理是一个很有挑战性的工作，数据流本身具有持续达到、速度快且规模巨大等特点，因此通常不会对所有的数据进行永久化存储，而且数据环境处在不断的变化之中，系统很难准确掌握整个数据的全貌。由于响应时间的要求，流处理的过程基本在内存中完成，其处理方式更多的依赖于在内存中设计巧妙的概要数据结构(Synopsis data structure)，内存容量是限制流处理模型的一个主要瓶颈。以 PCM(相变存储器)为代表的 SCM(Storage Class Memory，储存级内存)设备的出现或许可以使内存未来不再成为流处理模型的制约。

数据流的理论及技术研究已经有十几年的历史，目前仍旧是研究热点。于此同时很多实际系统也已开发和得到广泛的应用，比较代表性的开源系统如 Twitter 的 Storm、Yahoo 的 S4 以及 Linkedin 的 Kafka 等。

2.2.3 批处理

Google 公司在 2004 年提出的 MapReduce 编程模型是最具代表性的批处理模式。一个完整的 MapReduce 过程如图 2-5 所示。

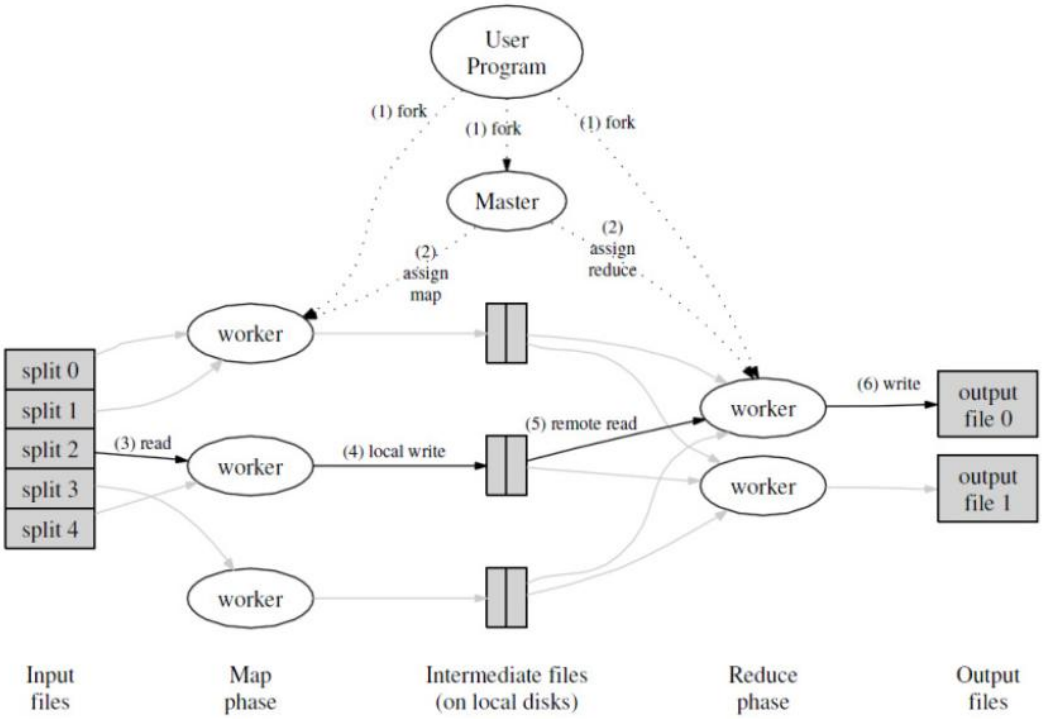

图 2-5 MapReduce 过程

MapReduce 模型首先将用户的原始数据源进行分块，然后分别交给不同的 Map 任务区处理。Map 任务从输入中解析出 Key/Value 对集合，然后对这些集合执行用户自行定义的 Map 函数得到中间结果，并将该结果写入本地硬盘。Reduce 任务从硬盘上读取数据之后，会根据 key 值进行排序，将具有相同 key 值的组织在一起。最后用户自定义的 Reduce 函数会作用于这些排好序的结果并输出最终结果。

从 MapReduce 的处理过程我们可以看出，MapReduce 的核心设计思想在于：1)将问题分而治之；2)把计算推到数据而不是把数据推到计算，有效地避免数据传输过程中产生的大量通讯开销。MapReduce 模型简单，且现实中很多问题都可用 MapReduce 模型来表示。因此该模型公开后，立刻受到极大的关注，并在生物信息学、文本挖掘等领域得到广泛的应用。无论是流处理还是批处理，都是大数据处理的可行思路。大数据的应用类型很多，在实际的大数据处理中，常常并不是简单的只使用其中的某一种，而是将二者结合起来。互联网是大数据最重要的来源之一，很多互联网公司根据处理时间的要求将自己的业务划分为在线(Online)、近线(Nearline)和离线(Offline)，比如著名的职业社交网站 Linkedin。这种划分方式是按处理所耗时间来划分的。其中在线的处理时间一般在秒级，甚至是毫秒级，因此通常采用上面所说的流处理。离线的处理时间可以以天为基本单位，基本采用批处理方式，这种方式可以最大限度地利用系统 I/O。近线的处理时间一般在分钟级或者是小时级，对其处理模

型并没有特别的要求，可以根据需求灵活选择。但在实际中多采用批处理模式。

2.3 大数据关键技术

如果将各种大数据的应用比作一辆辆“汽车”的话，支撑起这些“汽车”运行的“高速公路”就是云计算。正是云计算技术在数据存储、管理与分析等方面的支撑，才使得大数据有用武之地。在所有的“高速公路”中，Google 无疑是技术最为先进的一个。需求推动创新，面对海量的 Web 数据， Google 于 2006 年首先提出了云计算的概念。支撑 Google 内部各种大数据应用的正是其自行研发的一系列云计算技术和工具。难能可贵的是 Google 并未将这些技术完全封闭，而是以论文的形式逐步公开其实现。正是这些公开的论文，使得以 GFS、MapReduce、Bigtable 为代表的一系列大数据处理技术被广泛了解并得到应用，同时还催生出以 Hadoop 为代表的一系列云计算开源工具。云计算技术很多，但是通过 Google 云计算技术的介绍能够快速、完整地把握云计算技术的核心和精髓。这里以 Google 的相关技术介绍为主线，详细介绍 Google 以及其他众多学者和研究机构在大数据技术方面已有的一些工作。根据 Google 已公开的论文及相关资料，结合大数据处理的需求，可以看出 Google 的技术演化过程，如图 2-6 所示。

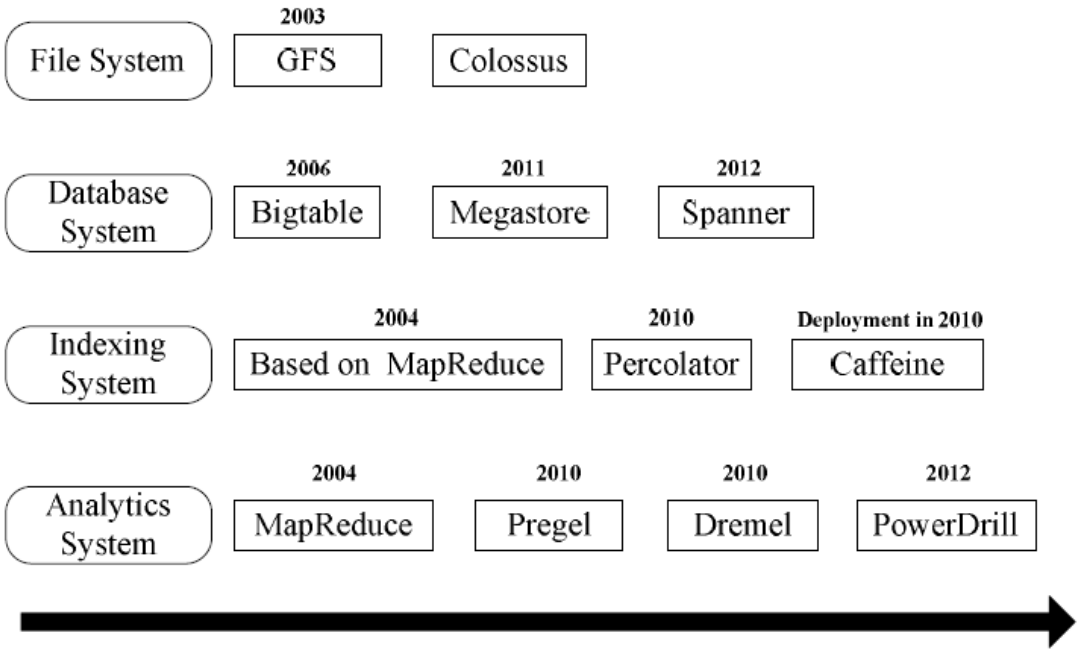

图 2-6 Google 的技术演化过程

2.3.1 文件系统

文件系统是支撑上层应用的基础。在 Google 之前，尚未有哪个公司面对过如此多的海量数据。因此，对于 Google 而言并没有完全成熟的存储方案可以直接使用。Google 认为系统组件失败是一种常态而不是异常，基于此思想，Google 自行设计开发了 Google 文件系统 GFS(Google File System)。GFS 是构建在大量廉价服务器之上的一个可扩展的分布式文件系统，GFS 主要针对文件较大，且读远大于写的应用场景，采用主从(Master-Slave)结构。

GFS 通过数据分块、追加更新(Append-Only)等方式实现了海量数据的高效存储。随着时间推移，GFS 的架构逐渐开始无法适应需求。Google 对 GFS 进行了重新的设计，该系统正式的名称为 Colossus，其中，GFS 的单点故障(指仅有一个主节点容易成为系统的瓶颈)、海量小文件的存储等问题在 Colossus 中均得到了解决。除了 Google，众多企业和学者也从不同方面对满足大数据存储需求的文件系统进行了详尽的研究。微软自行开发的 Cosmos 支撑着其搜索、广告等业务。HDFS 和 CloudStore 都是模仿 GFS 的开源实现。GFS 类的文件系统主要是针对较大文件设计的，而在图片存储等应用场景，文件系统主要存储海量小文件，此时 GFS 等文件系统因为频繁读取元数据等原因，效率很低。针对这种情况，Facebook 推出了专门针对海量小文件的文件系统 Haystack，通过多个逻辑文件共享同一个物理文件、增加缓存层、部分元数据加载到内存等方式有效的解决了 Facebook 海量图片存储问题。淘宝推出了类似的文件系统 TFS(Tao File System)，通过将小文件合并成大文件、文件名隐含部分元数据等方式实现了海量小文件的高效存储。FastDFS 针对小文件的优化类似于 TFS。

2.3.2 数据库系统

原始的数据存储在文件系统之中，但是，用户习惯通过数据库系统来存取文件。因为这样会屏蔽掉底层的细节，且方便数据管理。直接采用关系模型的分布式数据库并不能适应大数据时代的数据存储，主要因为：

1)**规模效应所带来的压力**。大数据时代的数据量远远超过单机所能容纳的数据量，因此必须采用分布式存储的方式。这就需要系统具有很好的扩展性，但这恰恰是传统数据库的弱势之一。因为传统的数据库产品对于性能的扩展更倾向于 Scale-Up(纵向扩展)的方式，而这种方式对于性能的增加速度远低于需要处理数据的增长速度，且性能提升存在上限。适应大数据的数据库系统应当具有良好的 Scale-Out(横向扩展)能力，而这种性能扩展方式恰恰是传

统数据库所不具备的。即便是性能最好的并行数据库产品其 Scale-Out 能力也相对有限。

2)**数据类型的多样化**。传统的数据库比较适合结构化数据的存储，但是数据的多样性是大数据时代的显著特征之一。这也就是意味着除了结构化数据，半结构化和非结构化数据也将是大数据时代的重要数据类型组成部分。如何高效地处理多种数据类型是大数据时代数据库技术面临的重要挑战之一。

3)**设计理念的冲突**。关系数据库追求的是“One size fits all”的目标，希望将用户从繁杂的数据管理中解脱出来，在面对不同的问题时不需要重新考虑数据管理问题，从而可以将重心转向其他的部分。但在大数据时代不同的应用领域在数据类型、数据处理方式以及数据处理时间的要求上有极大的差异。在实际的处理中几乎不可能有一种统一的数据存储方式能够应对所有场景。比如对于海量 Web 数据的处理就不可能和天文图像数据采取同样的处理方式。在这种情况下，很多公司开始尝试从“One size fits one”和“One size fits domain”的设计理念出发来研究新的数据管理方式，并产生了一系列非常有代表性的工作。

4)**数据库事务特性**。众所周知，关系数据库中事务的正确执行必须满足 ACID 特性，即原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)和持久性(Durability)。对于数据强一致性的严格要求使其在很多大数据场景中无法应用。这种情况下出现了新的 BASE 特性，即只要求满足 Basically Available(基本可用), Soft state(柔性状态)和 Eventually consistent(最终一致)。从分布式领域著名的 CAP 理论的角度来看，ACID 追求一致性 C，而 BASE 更加关注可用性 A。正是在事务处理过程中对于 ACID 特性的严格要求，使得关系型数据库的可扩展性极其有限。

面对这些挑战，以 Google 为代表的一批技术公司纷纷推出了自己的解决方案。Bigtable 是 Google 早期开发的数据库系统，它是一个多维稀疏排序表，由行和列组成，每个存储单元都有一个时间戳，形成三维结构。不同的时间对同一个数据单元的多个操作形成的数据的多个版本之间由时间戳来区分。除了 Bigtable，Amazon 的 Dynamo 和 Yahoo 的 PNUTS 也都是非常具有代表性的系统。Dynamo 综合使用了键/值存储、改进的分布式哈希表(DHT)、向量时钟(Vector Clock)等技术实现了一个完全的分布式、去中心化的高可用系统。PNUTS 是一个分布式的数据库，在设计上使用弱一致性来达到高可用性的目标，主要的服务对象是相对较小的记录，比如在线的大量单个记录或者小范围记录集合的读和写访问。不适合存储大文件、流媒体等。Bigtable、Dynamo、PNUTS 等的成功促使人们开始对关系数据库进行反思，由此产生了一批未采用关系模型的数据库，这些方案现在被统一的称为 NoSQL(Not Only SQL)。NoSQL 并没有一个准确的定义，但一般认为 NoSQL 数据库应当具有以下的特征：

模式自由(schema-free)、支持简易备份(easy replication support)、简单的应用程序接口(simple API)、最终一致性(或者说支持 BASE 特性，不支持 ACID)、支持海量数据(Huge amount of data)。

Bigtable 的模型简单，但是，相较传统的关系数据库其支持的功能非常有限，不支持 ACID 特性。因此，Google 开发了 Megastore 系统，虽然其底层数据存储依赖 Bigtable，但是它实现了类似 RDBMS 的数据模型，同时提供数据的强一致性解决方案。Megastore 将数据进行细粒度的分区，数据更新会在机房间进行同步复制。Spanner 是已知的 Google 的最新的数据库系统，Google 在 OSDI2012 上公开了 Spanner 的实现。Spanner 是第一个可以实现全球规模扩展(Global Scale)并且支持外部一致的事务(support externally-consistent distributed transactions)的数据库。通过 GPS 和原子时钟(atomic clocks)技术，Spanner 实现了一个时间 API。借助该 API，数据中心之间的时间同步能够精确到 10ms 以内。Spanner 类似于 Bigtable，但是它具有层次性的目录结构以及细粒度的数据复制。对于数据中心之间不同操作会分别支持强一致性或弱一致性，且支持更多的自动操作。Spanner 的目标是控制一百万到一千万台服务器，最多包含大约 10 万亿目录和一千万亿字节的存储空间。另外，在 SIGMOD2012 上，Google 公开了用于其广告系统的新数据库产品 F1，作为一种混合型数据库，F1 融合兼有 Bigtable 的高扩展性以及 SQL 数据库的可用性和功能性。该产品的底层存储正是采用 Spanner，具有很多新的特性，包括全局分布式、同步跨数据中心复制、可视分片和数据移动、常规事务等。有些比较激进的观点认为“关系数据库已死”，但是，一般而言，关系数据库和 NoSQL 并不是矛盾的对立体，而是可以相互补充的、适用于不同应用场景的技术。例如，实际的互联网系统往往都是 ACID 和 BASE 两种系统的结合。近些年来，以 Spanner 为代表的若干新型数据库的出现，给数据存储带来了 SQL、NoSQL 之外的新思路。这种融合了一致性和可用性的 NewSQL 或许会是未来大数据存储新的发展方向。

2.3.3 索引和查询技术

数据查询是数据库最重要的应用之一。而索引则是解决数据查询问题的有效方案。就 Google 自身而言，索引的构建是提供搜索服务的关键部分。Google 最早的索引系统是利用 MapReduce 来更新的。根据更新频率进行层次划分，不同的层次对应不同的更新频率。每次需要批量更新索引，即使有些数据并未改变也需要处理掉。这种索引更新方式效率较低。

随后 Google 提出了 Percolator，这是一种增量式的索引更新器，每次更新不需要替换所

有的索引数据，效率大大提高。虽然不是所有的大数据应用都需要索引，但是这种增量计算的思想非常值得我们借鉴。Google 当前正在使用的索引系统为 Caffeine，构建在 Spanner 之上，采用 Percolator 更新索引。效率相较上一代索引系统而言有大幅度提高。

关系数据库也是利用对数据构建索引的方式较好地解决了数据查询的问题。不同的索引方案使得关系数据库可以满足不同场景的要求。索引的建立以及更新都会耗费较多的时间，在面对传统数据库的小数据量时这些时间和其所带来的查询便利性相比是可以接受的，但是这些复杂的索引方案基本无法直接应用到大数据之上。表 2-1 是对一些索引方案直接应用在 Facebook 上的性能估计。

表 2-1 一些索引方案直接应用在 Facebook 上的性能估计

Algorithms	Index Size for Facebook	Index Time for Facebook	Query Time on Facebook(s)
Ullmann[Ullmann 76]	-	-	>1000
VF2[CordellaFSV04]	-	-	>1000
RDF-3X[NeumannW10]	1T	>20 days	>48
BitMat[AtreCZH10]	2.4T	>20days	>269
Subdue[HolderCD94]	-	>67 years	-
SpiderMine[ZhuQLYHY11]	-	>3 years	-
R-Join[ChengYDYW08]	>175T	$>10^{15}$ years	>200
Distance-Join[ZouCO09]	>175T	$>10^{15}$ years	>4000
GraphQL[HeS08]	>13T(r=2)	>600 years	>2000
Zhao[ZhaoH10]	>12T(r=2)	>600 years	>600
GADDI[ZhangLY09]	$>2*10^5$T(L=4)	$>4*10^5$ years	>400

从上表可以看出不太可能将已有的成熟索引方案直接应用于大数据。NoSQL 数据库针对主键的查询效率一般较高，因此有关的研究集中在 NoSQL 数据库的多值查询优化上。针对 NoSQL 数据库上的查询优化研究主要有两种思路：

1）采用 MapReduce 并行技术优化多值查询：当利用 MapReduce 并行查询 NoSQL 数据库时，每个 MapTask 处理一部分的查询操作，通过实现多个部分之间的并行查询来提高多值查询的效率。此时每个部分的内部仍旧需要进行数据的全扫描。

2）采用索引技术优化多值查询：很多的研究工作尝试从添加多维索引的角度来加速 NoSQL 数据库的查询速度。

就已有方案来看，针对 NoSQL 数据库上的查询优化技术都并不成熟，仍有很多关键性问题亟待解决。

2.3.4 数据分析技术

数据分析是 Google 最核心的业务，每一次简单的网络点击背后都需要进行复杂的分析过程，因此 Google 对其分析系统进行不断地升级改造之中。MapReduce 是 Google 最早采用的计算模型，适用于批处理。图是真实社会中广泛存在的事物之间联系的一种有效表示手段，因此，对图的计算是一种常见的计算模式，而图计算会涉及到在相同数据上的不断更新以及大量的消息传递，如果采用 MapReduce 去实现，会产生大量不必要的序列化和反序列化开销。现有的图计算系统并不适用于 Google 的应用场景，因此 Google 设计并实现了 Pregel 图计算模型。Pregel 是 Google 继 MapReduce 之后提出的又一个计算模型，与 MapReduce 的离线批处理模式不同，它主要用于图的计算。该模型的核心思想源于著名的 BSP 计算模型。Dremel 是 Google 提出的一个适用于 Web 数据级别的交互式数据分析系统，通过结合列存储和多层次的查询树，Dremel 能够实现极短时间内的海量数据分析。Dremel 支持着 Google 内部的一些重要服务，比如 Google 的云端大数据分析平台 Big Query。Google 在 VLDB 2012 发表的文章中介绍了一个内部名称为 PowerDrill 的分析工具，PowerDrill 同样采用了列存储，且使用了压缩技术将尽可能多的数据装载进内存。PowerDrill 与 Dremel 均是 Google 的大数据分析工具，但是其关注的应用场景不同，实现技术也有很大差异。Dremel 主要用于多数据集的分析，而 PowerDrill 则主要应用于大数据量的核心数据集分析，数据集的种类相较于 Dremel 的应用场景会少很多。由于 PowerDrill 是设计用来处理少量的核心数据集，因此对数据处理速度要求极高，所以其数据应当尽可能的驻留在内存，而 Dremel 的数据则存储在磁盘中。除此之外，PowerDrill 与 Dremel 在数据模型、数据分区等方面都有明显的差别。从实际的执行效率来看，Dremel 可以在几秒内处理 PB 级的数据查询，而 PowerDrill 则可以在 30 至 40 秒里处理 7820 亿个单元格的数据，处理速度快于 Dremel。二者的应用场景不同，可以相互补充。

微软提出了一个类似 MapReduce 的数据处理模型，称之为 Dryad，Dryad 模型主要用来构建支持有向无环图(Directed Acycline Graph，DAG)类型数据流的并行程序。Cascading 通过对 Hadoop MapReduce API 的封装，支持有向无环图类型的应用。Sector/sphere 可以视为一种流式的 MapReduce，它由分布式文件系统 Sector 和并行计算框架 sphere 组成。Nephele/PACTs 则包括 PACTs(Parallelization Contracts)编程模型和并行计算引擎 Nephele。MapReduce 模型基本成为了批处理类应用的标准处理模型，很多应用开始尝试利用 MapReduce 加速其数据处理。

实时数据处理是大数据分析的一个核心需求。很多研究工作正是围绕这一需求展开的。前面介绍了大数据处理的两种基本模式，而在实时处理的模式选择中，主要有三种思路：

1) 采用流处理模式。虽然流处理模式天然适合实时处理系统，但其适用领域相对有限。流处理模型的应用主要集中在实时统计系统、在线状态监控等。

2) 采用批处理模式。近几年来，利用批处理模型开发实时系统已经成为研究热点并取得了很多成果。从增量计算的角度出发，Google 提出了增量处理系统 Percolator，微软则提出了 Nectar 和 DryadInc。三者均实现了大规模数据的增量计算。

3) 二者的融合。有不少研究人员尝试将流处理和批处理模式进行融合，主要思路是利用 MapReduce 模型实现流处理。

2.4 大数据处理工具

关系数据库在很长的时间里成为数据管理的最佳选择，但是在大数据时代，数据管理、分析等的需求多样化使得关系数据库在很多场景不再适用。这里对现今主流的大数据处理工具进行一个简单的归纳和总结。

Hadoop 是目前最为流行的大数据处理平台。Hadoop 最先是 Doug Cutting 模仿 GFS、MapReduce 实现的一个云计算开源平台，后贡献给 Apache。Hadoop 已经发展成为包括文件系统(HDFS)、数据库(HBase、Cassandra)、数据处理(MapReduce)等功能模块在内的完整生态系统(Ecosystem)。某种程度上可以说 Hadoop 已经成为了大数据处理工具事实上的标准。

对 Hadoop 改进并将其应用于各种场景的大数据处理已经成为新的研究热点，主要的研究成果集中在对 Hadoop 平台性能的改进、高效的查询处理、索引构建和使用、在 Hadoop 之上构建数据仓库、Hadoop 和数据库系统的连接、数据挖掘、推荐系统等。

除了 Hadoop，还有很多针对大数据的处理工具。这些工具有些是完整的处理平台，有些则是专门针对特定的大数据处理应用。表 2-2 归纳总结了现今一些主流的处理平台和工具，这些平台和工具或是已经投入商业使用，或是开源软件。在已经投入商业使用的产品中，绝大部分也是在 Hadoop 基础上进行功能扩展，或者提供与 Hadoop 的数据接口。

表 2-2 大数据处理平台和工具

Category		Examples
Platform	Local	Hadoop、MapR、Cloudera、Hortonworks、InfoSphere BigInsights、ASTERIX
	Cloud	AWS、Google Compute Engine、Azure
Database	SQL	Greenplum、Aster Data、Vertica
	NoSQL	HBase、Cassandra、MongoDB、Redis
	NewSQL	Spanner、Megastore、F1
Data Warehouse		Hive、HadoopDB、Hadapt
Data Processing	Batch	MapReduce、Dryad
	Stream	Storm、S4、Kafka
Query Language		HiveQL、Pig Latin、DryadLINQ、MRQL、SCOPE
Statistic and Machine Learning		Mahout、Weka、R
Log Processing		Splunk、Loggly

2.5 大数据时代面临的新挑战

大数据时代的数据存在着如下几个特点：多源异构、分布广泛、动态增长、先有数据后有模式。正是这些与传统数据管理迥然不同的特点，使得大数据时代的数据管理面临着新的挑战，下面会对其中的主要挑战进行详细分析。

2.5.1 大数据集成

数据的广泛存在性使得数据越来越多的散布于不同的数据管理系统中，为了便于进行数据分析需要进行数据的集成。数据集成看起来并不是一个新的问题，但是大数据时代的数据集成却有了新的需求，因此也面临着新的挑战。

（1）广泛的异构性。传统的数据集成中也会面对数据异构的问题，但是在大数据时代这种异构性出现了新的变化。主要体现在：1）数据类型从以结构化数据为主转向结构化、半结构化、非结构化三者的融合。2）数据产生方式的多样性带来的数据源变化。传统的数据主要产生于服务器或者是个人电脑，这些设备位置相对固定。随着移动终端的快速发展，手机、平板电脑、GPS 等产生的数据量呈现爆炸式增长，且产生的数据带有很明显的时空特性。3）数据存储方式的变化。传统数据主要存储在关系数据库中，但越来越多的数据开始采用新的数据存储方式来应对数据爆炸，比如存储在 Hadoop 的 HDFS 中。这就必然要求在集成的过程中进行数据转换，而这种转换的过程是非常复杂和难以管理的。

（2）数据质量。数据量大不一定就代表信息量或者数据价值的增大，相反很多时候意味着信息垃圾的泛滥。一方面很难有单个系统能够容纳下从不同数据源集成的海量数据；另

一方面如果在集成的过程中仅仅简单地将所有数据聚集在一起而不做任何数据清洗，会使得过多的无用数据干扰后续的数据分析过程。大数据时代的数据清洗过程必须更加谨慎，因为相对细微的有用信息混杂在庞大的数据量中。如果信息清洗的粒度过细，很容易将有用的信息过滤掉。清洗粒度过粗，又无法达到真正的清洗效果，因此在质与量之间需要进行仔细的考量和权衡。

2.5.2 大数据分析

传统意义上的数据分析(analysis)主要针对结构化数据展开，且已经形成了一整套行之有效的分析体系。首先利用数据库来存储结构化数据，在此基础上构建数据仓库，根据需要构建数据立方体进行联机分析处理 (OLAP, Online Analytical Processing)，可以进行多个维度的下钻(Drill-down)或上卷(Roll-up)操作。对于从数据中提炼更深层次的知识的需求促使数据挖掘技术的产生，并发明了聚类、关联分析等一系列在实践中行之有效的方法。这一整套处理流程在处理相对较少的结构化数据时极为高效。但是，随着大数据时代的到来，半结构化和非结构化数据量的迅猛增长，给传统的分析技术带来了巨大的冲击和挑战，主要体现在：

（1）数据处理的实时性(Timeliness)。随着时间的流逝，数据中所蕴含的知识价值往往也在衰减，因此很多领域对于数据的实时处理有需求。随着大数据时代的到来，更多应用场景的数据分析从离线(offline)转向了在线(online)，开始出现实时处理的需求，比如 KDD 2012 最佳论文所探讨的实时广告竞价问题。大数据时代的数据实时处理面临着一些新的挑战，主要体现在数据处理模式的选择及改进。在实时处理的模式选择中，主要有三种思路：即流处理模式、批处理模式以及二者的融合。虽然已有的研究成果很多，但是仍未有一个通用的大数据实时处理框架。各种工具实现实时处理的方法不一，支持的应用类型都相对有限，这导致实际应用中往往需要根据自己的业务需求和应用场景对现有的这些技术和工具进行改造才能满足要求。

（2）动态变化环境中索引的设计。关系数据库中的索引能够加速查询速率，但是传统的数据管理中模式基本不会发生变化，因此在其上构建索引主要考虑的是索引创建、更新等的效率。大数据时代的数据模式随着数据量的不断变化可能会处于不断的变化之中，这就要求索引结构的设计简单、高效，能够在数据模式发生变化时很快的进行调整来适应。目前存在一些通过在 NoSQL 数据库上构建索引来应对大数据挑战的一些方案，但总的来说，这些方案基本都有特定的应用场景，且这些场景的数据模式不太会发生变化。在数据模式变更的

假设前提下设计新的索引方案将是大数据时代的主要挑战之一。

（3）先验知识的缺乏。传统分析主要针对结构化数据展开，这些数据在以关系模型进行存储的同时就隐含了这些数据内部关系等先验知识。比如我们知道所要分析的对象会有哪些属性，通过属性我们又能大致了解其可能的取值范围等。这些知识使得我们在数据分析之前就已经对数据有了一定的理解。而在面对大数据分析时，一方面是半结构化和非结构化数据的存在，这些数据很难以类似结构化数据的方式构建出其内部的正式关系；另一方面很多数据以流的形式源源不断的到来，这些需要实时处理的数据很难有足够的时间去建立先验知识。

2.5.3 大数据隐私问题

隐私问题由来已久，计算机的出现使得越来越多的数据以数字化的形式存储在电脑中，互联网的发展则使数据更加容易产生和传播，数据隐私问题越来越严重。

（1）隐性的数据暴露。很多时候人们有意识地将自己的行为隐藏起来，试图达到隐私保护的目的。但是互联网，尤其是社交网络的出现，使得人们在不同的地点产生越来越多的数据足迹。这种数据具有累积性和关联性，单个地点的信息可能不会暴露用户的隐私，但是如果有办法将某个人的很多行为从不同的独立地点聚集在一起时，他的隐私就很可能会暴露，因为有关他的信息已经足够多了，这种隐性的数据暴露往往是个人无法预知和控制的。从技术层面来说，可以通过数据抽取和集成来实现用户隐私的获取。而在现实中通过所谓的“人肉搜索”的方式往往能更快速、准确的得到结果，这种人肉搜索的方式实质就是众包(Crowd sourcing)。大数据时代的隐私保护面临着技术和人力层面的双重考验。

（2）数据公开与隐私保护的矛盾。如果仅仅为了保护隐私就将所有的数据都加以隐藏，那么数据的价值根本无法体现。数据公开是非常有必要的，政府可以从公开的数据中来了解整个国民经济社会的运行，以便更好地指导社会的运转。企业则可以从公开的数据中了解客户的行为，从而推出针对性的产品和服务，最大化其利益。研究者则可以利用公开的数据，从社会、经济、技术等不同的角度来进行研究。因此大数据时代的隐私性主要体现在不暴露用户敏感信息的前提下进行有效的数据挖掘，这有别于传统的信息安全领域更加关注文件的私密性等安全属性。统计数据库数据研究中最早开展数据隐私性技术方面的研究，近年来逐渐成为相关领域的研究热点。保护隐私的数据挖掘(privacy preserving data mining)这一概念，很多学者开始致力于这方面的研究。主要集中于研究新型的数据发布技术，尝试在尽可能少

损失数据信息的同时最大化的隐藏用户隐私。但是数据信息量和隐私之间是有矛盾的，因此尚未出现非常好的解决办法。Dwork 在 2006 年提出了新的差分隐私(Differential Privacy)方法。差分隐私保护技术可能是解决大数据中隐私保护问题的一个方向，但是这项技术离实际应用还很远。

（3）数据动态性。大数据时代数据的快速变化除了要求有新的数据处理技术应对之外，也给隐私保护带来了新的挑战。现有隐私保护技术主要基于静态数据集，而在现实中数据模式和数据内容时刻都在发生着变化。因此在这种更加复杂的环境下实现对动态数据的利用和隐私保护将更具挑战。

2.5.4 大数据能耗问题

在能源价格上涨、数据中心存储规模不断扩大的今天，高能耗已逐渐成为制约大数据快速发展的一个主要瓶颈。从小型集群到大规模数据中心都面临着降低能耗的问题，但是尚未引起足够多的重视，相关的研究成果也较少。在大数据管理系统中，能耗主要由两大部分组成：硬件能耗和软件能耗，二者之中又以硬件能耗为主。理想状态下，整个大数据管理系统的能耗应该和系统利用率成正比。但是实际情况并不像预期情况，系统利用率为 0 的时候仍然有能量消耗。针对这个问题，《纽约时报》和麦肯锡经过一年的联合调查，最终在《纽约时报》上发表文章《Power, Pollution and the Internet》。调查显示 Google 数据中心年耗电量约为 300 万瓦，而 Facebook 则在 60 万瓦左右。最令人惊讶的是在这些巨大的能耗中，只有 6%-12%的能量被用来响应用户的查询并进行计算。绝大部分的电能用以确保服务器处于闲置状态，以应对突如其来的网络流量高峰，这种类型的功耗最高可以占到数据中心所有能耗的 80%。从已有的一些研究成果来看，可以考虑以下两个方面来改善大数据能耗问题：

（1）采用新型低功耗硬件。从纽约时报的调查中可以知道绝大部分的能量都耗费在磁盘上。在空闲的状态下，传统的磁盘仍然具有很高的能耗，并且随着系统利用率的提高，能耗也在逐渐升高。新型非易失存储器件的出现，给大数据管理系统带来的新的希望。闪存、PCM 等新型存储硬件具有低能耗的特性。虽然随着系统利用率的提高，闪存、PCM 等的能耗也有所升高，但是其总体能耗仍远远低于传统磁盘。

（2）引入可再生的新能源。数据中心所使用的电能绝大部分都是从不可再生的能源中产生的。如果能够在大数据存储和处理中引入诸如太阳能、风能之类的可再生能源，将在很大程度上缓解能耗问题。这方面的工作很少，有研究人员探讨了如何利用太阳能构建一个绿

色环保的数据库。

2.5.5 大数据处理与硬件的协同

硬件的快速升级换代有力的促进了大数据的发展，但是这也在一定程度上造成了大量不同架构硬件共存的局面。日益复杂的硬件环境给大数据管理带来的主要挑战有：

（1）硬件异构性带来的大数据处理难题。整个数据中心（集群）内部不同机器之间的性能会存在着明显的差别，因为不同时期购入的不同厂商的服务器在 IOPS、CPU 处理速度等性能方面会有很大的差异。这就导致了硬件环境的异构性(Heterogeneous)，而这种异构性会给大数据的处理带来诸多问题。一个典型的例子就是 MapReduce 任务过程中，其总的处理时间很大程度上取决于 Map 过程中处理时间最长的节点。如果集群中硬件的性能差异过大，则会导致大量的计算时间浪费在性能较好的服务器等待性能较差的服务器上。这种情况下服务器的线性增长并不一定会带来计算能力的线性增长，因为“木桶效应”制约了整个集群的性能。一般的解决方案是考虑硬件异构的环境下将不同计算强度的任务智能的分配给计算能力不同的服务器，但是当这种异构环境的规模扩展到数以万计的集群时问题将变得极为复杂。

（2）新硬件给大数据处理带来的变革。所有的软件系统都是构建在传统的计算机体系结构之上，即 CPU-内存-硬盘三级结构。CPU 的发展一直遵循着摩尔定律，且其架构已经从单核转入多核。因此需要深入研究如何让软件更好的利用 CPU 多核心之间的并发机制。由于机械特性的限制，基于磁性介质的硬盘（Hard Disk Drive, HDD）的读写速率在过去几十年中提升不大，而且未来也不太可能出现革命性的提升。基于闪存的固态硬盘(Solid State Disk,SSD)的出现从硬件层为存储系统结构的革新提供了支持，为计算机存储技术的发展和存储能效的提高带来了新的契机。SSD 具有很多优良特性，主要包括极高的读写性能、抗震性、低功耗、体积小等，因此正得到越来越广泛的应用。但是直接将 SSD 应用到现有的软件上并不一定会带来软件性能的大幅提升。Sang-Won Lee 等人的研究表明虽然 SSD 的读写速率是 HDD 的 60~150 倍，基于 SSD 的数据库系统的查询时间却仅仅提升了不到 10 倍。二者之间的巨大差距主要是由 SSD 的一些特性造成的，这些特性包括：SSD 写前擦除特性导致的读写操作代价不对称、SSD 存储芯片的擦除次数有限等。软件设计之时必须仔细考虑这些特性才能够充分利用 SSD 的优良特性。与大容量磁盘和磁盘阵列相比，固态硬盘的存储容量相对较低，单位容量的价格远高于磁盘。且不同类型的固态硬盘产品性能差异较大，

将固态硬盘直接替换磁盘应用到现有的存储体系中难以充分发挥其性能。因此现阶段可以考虑通过构建 HDD 和 SSD 的混合存储系统来解决大数据处理问题。当前混合存储系统的实现主要有三种思路：HDD 作为内存的扩展充当 SSD 写缓冲；HDD 和 SSD 同做二级存储；SSD 用作内存的扩展充当 HDD 读写缓冲。国外的 Google、Facebook，国内的百度、淘宝等公司已经开始在实际运营环境中大规模的使用混合存储系统来提升整体性能。在这三级结构之中，内存的发展处于一个相对缓慢的阶段，一直没有出现革命性的变化。构建任何一个软件系统都会假设内存是一个容量有限的易失结构体。随着以 PCM 为代表的 SCM 的出现，未来的内存极有可能会兼具现在内存和磁盘的双重特性，即处理速度极快且非易失。虽然 PCM 尚未有可以大规模量产的产品推出，但是各大主流厂商都对其非常重视，三星电子在 2012 年国际固态电路会议(ISSCC 2012)上发表了采用 20nm 工艺制程的容量为 8G 的 PCM 元件。一旦 PCM 能够大规模的投入使用，必将给现有的大数据处理带来一场根本性的变革。譬如前面提到的流处理模式就可以不再将内存的大小限制作为算法设计过程中的一个主要考虑因素。

2.5.6 大数据管理易用性问题

从数据集成到数据分析，直到最后的数据解释，易用性应当贯穿整个大数据的流程。易用性的挑战突出体现在两个方面：首先大数据时代的数据量大，分析更复杂，得到的结果形式更加的多样化，其复杂程度已经远远超出传统的关系数据库。其次大数据已经广泛渗透到人们生活的各个方面，很多行业都开始有了大数据分析的需求。但是这些行业的绝大部分从业者都不是数据分析的专家，在复杂的大数据工具面前，他们只是初级的使用者(Naïve Users)。复杂的分析过程和难以理解的分析结果限制了他们从大数据中获取知识的能力。这两个原因导致易用性成为大数据时代软件工具设计的一个巨大挑战。关于大数据易用性的研究仍处于一个起步阶段。从设计学的角度来看易用性表现为易见(Easy to discover)、易学(Easyto learn)和易用 (Easy to use)。要想达到易用性，需要关注以下三个基本原则：

（1）可视化原则(Visibility)。可视性要求用户在见到产品时就能够大致了解其初步的使用方法，最终的结果也要能够清晰的展现出来。未来如何实现更多大数据处理方法和工具的简易化和自动化将是一个很大的挑战。除了功能设计之外，最终结果的展示也要充分体现可视化的原则。可视化技术是最佳的结果展示方式之一，通过清晰的图形图像展示直观地反映出最终结果。但是超大规模的可视化却面临着诸多挑战，主要有：原位分析、用户界面与交

互设计、大数据可视化、数据库与存储、算法、数据移动、传输和网络架构、不确定性的量化、并行化、面向领域与开发的库、框架以及工具、社会、社区以及政府参与等。

（2）匹配原则(Mapping)。人的认知中会利用现有的经验来考虑新的工具的使用。譬如一提到数据库，了解的人都会想到使用 SQL 语言来执行数据查询。在新工具的设计过程中尽可能将人们已有的经验知识考虑进去，会使得新工具非常便于使用，这就是所谓的匹配原则。MapReduce 模型虽然将复杂的大数据处理过程简化为 Map 和 Reduce 的过程，但是具体的 Map 和 Reduce 函数仍需要用户自己编写，这对于绝大部分没有编程经验的用户而言仍过于复杂。如何将新的大数据处理技术和人们已经习惯的处理技术和方法进行匹配将是未来大数据易用性的一个巨大挑战。这方面现在已经有了些初步的研究工作。针对 MapReduce 技术缺乏类似 SQL 标准语言的弱点，研究人员开发出更高层的语言和系统。典型代表有 Hadoop 的 HiveQL 和 Pig Latin、Google 的 Sawzall、微软的 SCOPE 和 DryadLINQ 以及 MRQL 等。

（3）反馈原则(Feedback)。带有反馈的设计使得人们能够随时掌握自己的操作进程。进度条就是一个体现反馈原则的经典例子。大数据领域关于这方面的工作较少，有部分学者开始关注 MapReduce 程序执行进程的估计。传统的软件工程领域，程序出现问题之后有比较成熟的调试工具可以对错误的程序进行交互式的调试，相对容易找到错误的根源。但是大数据时代很多工具其内部结构复杂，对于普通用户而言这些工具近似于黑盒(black box)，调试过程复杂，缺少反馈性。如果未来能够在大数据的处理中大范围的引入人机交互技术，使得人们能够较完整的参与整个分析过程，会有效的提高用户的反馈感，在很大程度上提高易用性。

满足三个基本原则的设计就能够达到良好的易用性。从技术层面来看，可视化、人机交互以及数据起源技术都可以有效的提升易用性。而在这些技术的背后，海量元数据管理的问题是需要我们特别关注的一个问题。元数据是关于数据的数据，数据之间的关联关系以及数据本身的一些属性大都是靠元数据来表示的。可视化技术离不开元数据的支持，因为如果无法准确的表征出数据之间的关系，就无法对数据进行可视化的展示。数据起源技术更是离不开元数据管理技术。因为数据起源需要利用元数据来记录数据之间包括因果关系在内的各种复杂关系，并通过这些信息来进行相关的推断。如何在大规模存储系统中实现海量元数据的高效管理将会对大数据的易用性产生重要影响。

2.5.7 性能测试基准

关系数据库产品的成功离不开以 TPC 系列为代表的测试基准的产生。正是有了这些测试基准，才能够准确的衡量不同数据库产品的性能，并对其存在的问题进行改进。目前尚未有针对大数据管理的测试基准，构建大数据测试基准面临的主要挑战有：

（1）系统复杂度高。大数据管理系统的类型非常多，很多公司针对自己的应用场景设计了相应的数据库产品。这些产品的功能模块各异，很难用一个统一的模型来对所有的大数据产品进行建模。

（2）用户案例的多样性。测试基准需要定义一系列具有代表性的用户行为，但是大数据的数据类型广泛，应用场景也不尽相同，很难从中提取出具有代表性的用户行为。

（3）数据规模庞大。这会带来了两方面的挑战。首先数据规模过大使得数据重现非常困难，代价很大。其次在传统的 TPC 系列测试中，测试系统的规模往往大于实际客户使用的数据集，因此测试的结果可以准确的代表系统的实际性能。但是在大数据时代，用户实际使用系统的数据规模往往大于测试系统的数据规模，因此能否用小规模数据的测试基准来代表实际产品的性能是目前面临的一个挑战。数据重现的问题可以尝试利用一定的方法来去产生测试样例，而不是选择下载某个实际的测试数据集。但是这又涉及到如何使产生的数据集能真实反映原始数据集的问题。

（4）系统的快速演变。传统的关系数据库其系统架构一般比较稳定，但是大数据时代的系统为了适应数据规模的不断增长和性能要求的不断提升，必须不断的进行升级，这使得测试基准得到的测试结果很快就不能反映系统当前的实际性能。

（5）重新构建还是复用现有的测试基准。如果能够在现有的测试基准中选择合适的进行扩展的话，那么将极大减少构建新的大数据测试基准的工作量。可能的候选测试标准有 SWIM(Statistical Workload Injector for MapReduce)、MRBS、Hadoop 自带的 GridMix、TPC-DS、YCSB++等。

现在已经开始有工作尝试构建大数据的测试基准，比如一些针对大数据测试基准的会议 WBDB 2012、TPCTC 2012 等。但是也有观点认为当前讨论大数据测试基准的构建为时尚早。Yanpei Chen 等通过对 7 个应用 MapReduce 技术的实际产品的负载进行了跟踪和分析，认为现在根本无法确定大数据时代的典型用户案例。因此从这个角度来看并不适合构建大数据的测试基准，还有很多基础性的问题亟待解决。

总的来说，构建大数据的测试基准是有必要的。但是面临的挑战非常多，要想构建一个

类似 TPC 的公认的测试标准难度很大。

本章小结

本章内容首先介绍了大数据处理的基本流程和大数据处理模型，接着介绍了大数据的关键技术，其中，云计算是大数据的基础平台和支撑技术，本章以 Google 的相关技术为主线，详细介绍 Google 以及其他众多学者和研究机构在大数据技术方面已有的一些工作，包括文件系统、数据库系统、索引和查询技术、数据分析技术等；接下来，介绍了大数据处理平台和工具，就目前技术发展现状而言，Hadoop 已经成为了大数据处理工具事实上的标准。最后，介绍大数据时代面临的新挑战，包括大数据集成、大数据分析、大数据隐私问题、大数据能耗问题、大数据处理与硬件的协同、大数据管理易用性问题以及性能测试基准。

参考文献

[1]孟小峰, 慈祥. 大数据管理：概念、技术与挑战. 计算机学报, 2013 年第 8 期.

（注：本章绝大多数内容来自上面的参考文献）

第 3 章 Hadoop

Hadoop 是一个开源的、可运行于大规模集群上的分布式并行编程框架，它实现了 Map/Reduce 计算模型。借助于 Hadoop，程序员可以轻松地编写分布式并行程序，将其运行于计算机集群上，完成海量数据的计算。

本章介绍 Hadoop 相关知识，内容要点如下：

- Hadoop 概述
- Hadoop 发展简史
- Hadoop 的功能与作用
- 为什么不用关系型数据库管理系统
- Hadoop 的优点
- Hadoop 的应用现状和发展趋势
- Hadoop 项目及其结构
- Hadoop 的体系结构
- Hadoop 与分布式开发
- Hadoop 应用案例

3.1 Hadoop 概述

Hadoop 是 Apache 软件基金会旗下的一个开源分布式计算平台。以 Hadoop 分布式文件系统（HDFS，Hadoop Distributed File System）和 MapReduce（Google MapReduce 的开源实现）为核心的 Hadoop，为用户提供了系统底层细节透明的分布式基础架构。HDFS 的高容错性、高伸缩性等优点允许用户将 Hadoop 部署在低廉的硬件上，形成分布式系统；MapReduce 分布式编程模型允许用户在不了解分布式系统底层细节的情况下开发并行应用程序。所以，用户可以利用 Hadoop 轻松地组织计算机资源，从而搭建自己的分布式计算平台，并且可以充分利用集群的计算和存储能力，完成海量数据的处理（如图 3-1 所示）。

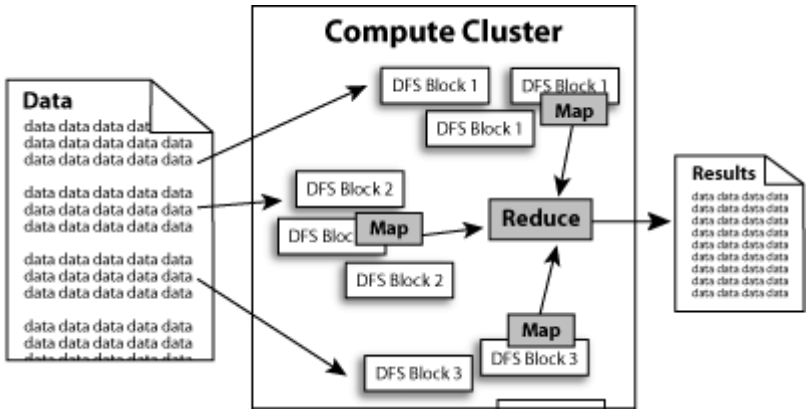

图 3-1 Hadoop 计算过程示意图

Hadoop 被公认是一套行业大数据标准开源软件，在分布式环境下提供了海量数据的处理能力。几乎所有主流厂商都围绕 Hadoop 开发工具、开源软件、商业化工具和技术服务。2013 年，大型 IT 公司，如 EMC、Microsoft、Intel、Teradata、Cisco 都明显增加了 Hadoop 方面的投入，Teradata 还公开展示了一个一体机；另一方面，创业型 Hadoop 公司层出不穷，如 Sqrrl、Wandisco、GridGain、InMobi 等等，都推出了开源的或者商用的软件。

3.2 Hadoop 发展简史

Hadoop 是 Doug Cutting——Apache Lucene 创始人——开发的使用广泛的文本搜索库。Hadoop 起源于 Apache Nutch，后者是一个开源的网络搜索引擎，本身也是 Lucene 项目的一部分。

Hadoop 这个名字不是一个缩写，它是一个虚构的名字。该项目的创建者，Doug Cutting 如此解释 Hadoop 的得名：“这个名字是我孩子给一头吃饱了的棕黄色大象命名的。我的命名标准就是简短，容易发音和拼写，没有太多的意义，并且不会被用于别处。小孩子是这方面的高手。Google 就是由小孩命名的”。

Hadoop 及其子项目和后继模块所使用的名字往往也与其功能不相关，经常用一头大象和其他动物主题（例如：Pig）。较小的各个组成部分给与更多描述性（因此也更俗）的名称。这是一个很好的原则，因为它意味着可以大致从其名字猜测其功能，例如，jobtracker 的任务就是跟踪 MapReduce 作业。

从头开始构建一个网络搜索引擎是一个雄心勃勃的目标，不只是要编写一个复杂的、能够抓取和索引网站的软件，还需要面临着没有专业运行团队支持运行它的挑战，因为它有那么多独立部件。同样昂贵的还有：据 Mike Cafarella 和 Doug Cutting 估计，一个支持此 10

亿页的索引需要价值约 50 万美元的硬件投入，每月运行费用还需要 3 万美元。不过，他们相信这是一个有价值的目标，因为这会开放并最终使搜索引擎算法普及化。

Nutch 项目开始于 2002 年，一个可工作的抓取工具和搜索系统很快浮出水面。但他们意识到，他们的架构将无法扩展到拥有数十亿网页的网络。在 2003 年发表的一篇描述 Google 分布式文件系统（Google File System，简称 GFS）的论文为他们提供了及时的帮助，文中称 Google 正在使用此文件系统。GFS 或类似的东西，可以解决他们在网络抓取和索引过程中产生的大量的文件存储需求。具体而言，GFS 会省掉管理所花的时间，如管理存储节点。在 2004 年，他们开始写一个开放源码的应用，即 Nutch 的分布式文件系统（NDFS）。

2004 年，Google 发表了论文，向全世界介绍了 MapReduce。2005 年初，Nutch 的开发者在 Nutch 上有了一个可工作的 MapReduce 应用，到当年年中，所有主要的 Nutch 算法被移植到使用 MapReduce 和 NDFS 来运行。

Nutch 中的 NDFS 和 MapReduce 实现的应用远不只是搜索领域，在 2006 年 2 月，他们从 Nutch 转移出来成为一个独立的 Lucene 子项目，成为 Hadoop。大约在同一时间，Doug Cutting 加入雅虎，Yahoo 提供一个专门的团队和资源将 Hadoop 发展成一个可在网络上运行的系统。在 2008 年 2 月，雅虎宣布其搜索引擎产品部署在一个拥有 1 万个内核的 Hadoop 集群上。

2008 年 1 月，Hadoop 已成为 Apache 顶级项目，证明它是成功的，是一个多样化、活跃的社区。通过这次机会，Hadoop 成功地被雅虎之外的很多公司应用，如 Last.fm、Facebook 和《纽约时报》。

有一个良好的宣传范例，《纽约时报》使用亚马逊的 EC2 云计算将 4TB 的报纸扫描文档压缩，转换为用于 Web 的 PDF 文件。这个过程历时不到 24 小时，使用 100 台机器运行，如果不结合亚马逊的按小时付费的模式（即允许《纽约时报》在很短的一段时间内访问大量机器）和 Hadoop 易于使用的并行程序设计模型，该项目很可能不会这么快开始启动。

2008 年 4 月，Hadoop 打破世界纪录，成为最快排序 1TB 数据的系统。运行在一个 910 节点的群集，Hadoop 在 209 秒内排序了 1TB 的数据（还不到三分半钟），击败了前一年的 297 秒冠军。同年 11 月，谷歌在报告中声称，它的 MapReduce 实现执行 1TB 数据的排序只用了 68 秒。在 2009 年 5 月，有报道宣称 Yahoo 的团队使用 Hadoop 对 1TB 的数据进行排序只花了 62 秒时间。

Hadoop 大事记

- 2004 年——最初的版本（现在称为 HDFS 和 MapReduce）由 Doug Cutting 和 Mike Cafarella 开始实施。
- 2005 年 12 月——Nutch 移植到新的框架，Hadoop 在 20 个节点上稳定运行。
- 2006 年 1 月——Doug Cutting 加入雅虎。
- 2006 年 2 月——Apache Hadoop 项目正式启动以支持 MapReduce 和 HDFS 的独立发展。
- 2006 年 2 月——雅虎的网络计算团队采用 Hadoop。
- 2006 年 4 月——标准排序（10GB 每个节点）在 188 个节点上运行 47.9 个小时。
- 2006 年 5 月——雅虎建立了一个 300 个节点的 Hadoop 研究集群。
- 2006 年 5 月——标准排序在 500 个节点上运行 42 个小时（硬件配置比 4 月的更好）。
- 2006 年 11 月——研究集群增加到 600 个节点。
- 2006 年 12 月——标准排序在 20 个节点上运行 1.8 个小时，100 个节点 3.3 小时，500 个节点 5.2 小时，900 个节点 7.8 个小时。
- 2007 年 1 月——研究集群到达 900 个节点。
- 2007 年 4 月——研究集群达到两个 1000 个节点的集群。
- 2008 年 4 月——赢得世界最快 1TB 数据排序在 900 个节点上用时 209 秒。
- 2008 年 10 月——研究集群每天装载 10TB 的数据。
- 2009 年 3 月——17 个集群总共 24000 台机器。
- 2009 年 4 月——赢得每分钟排序，59 秒内排序 500GB（在 1400 个节点上）和 173 分钟内排序 100TB 数据（在 3400 个节点上）。

3.3 Hadoop 的功能与作用

我们为什么需要 Hadoop 呢？众所周知，现代社会的信息量增长速度极快，这些信息里又积累着大量的数据，其中包括个人数据和工业数据。预计到 2020 年，每年产生的数字信息将会有超过 1/3 的内容驻留在云平台中或借助云平台来处理。我们需要对这些数据进行分析和处理，以获取更多有价值的信息。那么我们如何高效地存储和管理这些数据，如何分析这些数据呢？这时可以选用 Hadoop 系统，它在处理这类问题时，采用了分布式存储方式，

提高了读写速度，并扩大了存储容量。采用 MapReduce 来整合分布式文件系统上的数据，可保证分析和处理数据的高效。与此同时，Hadoop 还采用存储冗余数据的方式保证了数据的安全性。

Hadoop 中 HDFS 的高容错特性以及它是基于 Java 语言开发的，这使得 Hadoop 可以部署在低廉的计算机集群中，同时不限于某个操作系统。Hadoop 中 HDFS 的数据管理能力，MapReduce 处理任务时的高效率，以及它的开源特性，使其在同类的分布式系统中大放异彩，并在众多行业和科研领域中被广泛采用。

3.4 为什么不用关系型数据库管理系统

为什么我们不能使用数据库加上更多磁盘来做大规模的批量分析呢？为什么需要 MapReduce？

这个问题的答案来自于磁盘驱动器的另一个发展趋势：寻址时间的提高速度远远慢于传输速率的提高速度。寻址就是将磁头移动到特定位置进行读写操作的工序，它的特点是磁盘操作有延迟，而传输速率对应于磁盘的带宽。

如果数据的访问模式受限于磁盘的寻址，势必会导致它花更长时间(相较于流)来读或写大部分数据。另一方面，在更新一小部分数据库记录的时候，传统的 B 树(关系型数据库中使用的一种数据结构，受限于执行查找的速度)效果很好。但在更新大部分数据库数据的时候，B 树的效率就没有 MapReduce 的效率高，因为它需要使用排序/合并来重建数据库。

在许多情况下，MapReduce 能够被视为一种 RDBMS(关系型数据库管理系统)的补充(两个系统之间的差异见表 3-1)。MapReduce 很适合处理那些需要分析整个数据集的问题（以批处理的方式），尤其是 Ad Hoc(自主或即时)分析，而 RDBMS 则适用于点查询和更新（其中，数据集已经被索引以提供低延迟的检索和短时间的少量数据更新）。MapReduce 适合数据被一次写入和多次读取的应用，而关系型数据库更适合持续更新的数据集。

表 3-1 关系型数据库和 MapReduce 的比较

	传统关系型数据库	MapReduce
数据大小	GB	PB
访问	交互型和批处理	批处理
更新	多次读写	一次写入多次读取
结构	静态模式	动态模式

集成度	高	低
伸缩性	非线性	线性

MapReduce 和关系型数据库之间的另一个区别是，它们操作的数据集中的结构化数据的数量。结构化数据是拥有准确定义的实体化数据，具有诸如数据库表定义的格式，符合特定的预定义模式，这就是 RDBMS 包括的内容。另一方面，半结构化数据比较宽松（比如 XML 文档），虽然可能有模式，但经常被忽略，所以它只能用作数据结构指南。非结构化数据没有什么特别的内部结构，例如纯文本或图像数据。MapReduce 对于非结构化或半结构化数据非常有效，因为它被设计为在处理时间内解释数据。换句话说：MapReduce 输入的键和值并不是数据固有的属性，它们是由分析数据的人来选择的。

关系型数据往往是规范的，以保持其完整性和删除冗余。但是，规范化会给 MapReduce 带来问题，因为它使读取记录成为一个非本地操作，并且 MapReduce 的核心假设之一就是，它可以进行(高速)流的读写。

Web 服务器日志是记录集的一个很好的非规范化例子(例如，客户端主机名每次都以全名来指定，即使同一客户端可能会出现很多次)，这也是 MapReduce 非常适合用于分析各种日志文件的原因之一。

MapReduce 是一种线性的可伸缩的编程模型。程序员编写两个函数——map 函数和 Reduce 函数——每一个都定义一个键/值对集映射到另一个。这些函数无视数据的大小或者它们正在使用的集群的特性，这样它们就可以原封不动地应用到小规模数据集或者大的数据集上。更重要的是，如果放入两倍的数据量，运行的时间会少于两倍。但是如果是两倍大小的集群，一个任务仍然只是和原来的一样快。这不是一般的 SQL 查询的效果。

随着时间的推移，关系型数据库和 MapReduce 之间的差异很可能变得模糊。一方面，关系型数据库都开始吸收 MapReduce 的一些思路(如 ASTER DATA 和 GreenPlum 的数据库)；另一方面，基于 MapReduce 的高级查询语言(如 Pig 和 Hive)使 MapReduce 的系统更接近传统的数据库编程人员。

3.5 Hadoop 的优点

Hadoop 是一个能够对大量数据进行分布式处理的软件框架，并且是以一种可靠、高效、可伸缩的方式进行处理的，它具有以下几个方面的优点：

- 高可靠性：因为它假设计算元素和存储会失败，因此它维护多个工作数据副本，确保能够针对失败的节点重新分布处理。
- 高效性：因为它以并行的方式工作，通过并行处理加快处理速度。Hadoop 还是可伸缩的，能够处理 PB 级数据。Hadoop 能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此其处理速度非常快。
- 高可扩展性：Hadoop 是在可用的计算机集簇间分配数据并完成计算任务的，这些集簇可以方便地扩展到数以千计的节点中。
- 高容错性：Hadoop 能够自动保存数据的多个副本，并且能够自动将失败的任务重新分配。
- Hadoop 成本低：依赖于廉价服务器，因此它的成本比较低，任何人都可以使用。
- 运行在 Linux 平台上：Hadoop 带有用 Java 语言编写的框架，因此运行在 Linux 生产平台上是非常理想的。
- 支持多种编程语言：Hadoop 上的应用程序也可以使用其他语言编写，比如 C++。

3.6 Hadoop 的应用现状和发展趋势

由于Hadoop优势突出，基于Hadoop的应用已经遍地开花，尤其是在互联网领域。Yahoo!通过集群运行 Hadoop，以支持广告系统和 Web 搜索的研究；Facebook 借助集群运行 Hadoop，以支持其数据分析和机器学习；百度则使用 Hadoop 进行搜索日志的分析和网页数据的挖掘工作；淘宝的 Hadoop 系统用于存储并处理电子商务交易的相关数据；中国移动研究院基于 Hadoop 的“大云”（BigCloud）系统，用于对数据进行分析和对外提供服务。

2008 年 2 月，Hadoop 最大贡献者的 Yahoo！构建了当时规模最大的 Hadoop 应用，它们在 2000 个节点上面执行了超过 1 万个 Hadoop 虚拟机器，来处理超过 5PB 的网页内容，分析大约 1 兆个网络连接之间的网页索引资料。这些网页索引资料压缩后超过 300TB。Yahoo！正是基于这些为用户提供了高质量的搜索服务。

Hadoop 目前已经取得了非常突出的成绩。随着互联网的发展，新的业务模式还将不断涌现，Hadoop 的应用也会从互联网领域向电信、电子商务、银行、生物制药等领域拓展。相信在未来，Hadoop 将会在更多的领域中扮演幕后英雄，为我们提供更加快捷优质的服务。

3.7 Hadoop 项目及其结构

Hadoop 有许多元素构成。最底部是 Hadoop Distributed File System（HDFS），它存储 Hadoop 集群中所有存储节点上的文件。HDFS 的上一层是 MapReduce 引擎，该引擎由 JobTrackers 和 TaskTrackers 组成。

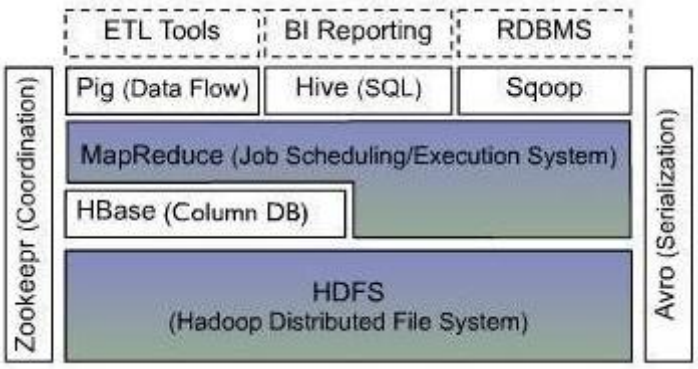

图 3-2 Hadoop 生态系统图

图 3-2 描述了 Hadoop 生态系统中的各层子系统，具体如下：

- **Avro**

Avro 是用于数据序列化的系统。它提供了丰富的数据结构类型、快速可压缩的二进制数据格式、存储持久性数据的文件集、远程调用 RPC 的功能和简单的动态语言集成功能。其中，代码生成器既不需要读写文件数据，也不需要使用或实现 RPC 协议，它只是一个可选的对静态类型语言的实现。Avro 系统依赖于模式（Schema），Avro 数据的读和写是在模式之下完成的。这样就可以减少写入数据的开销，提高序列化的速度并缩减其大小。同时，也可以方便动态脚本语言的使用，因为数据连同其模式都是自描述的。在 RPC 中，Avro 系统的客户端和服务端通过握手协议进行模式的交换。因此当客户端和服务端拥有彼此全部的模式时，不同模式下的相同命名字段、丢失字段和附加字段等信息的一致性问题就得到了很好的解决。

- **HDFS**

HDFS 是一种分布式文件系统，运行于大型商用机集群，HDFS 为 HBase 提供了高可靠性的底层存储支持；由于 HDFS 具有高容错性（fault-tolerant）的特点，所以可以设计部署在低廉（low-cost）的硬件上。它可以以很高的高吞吐率（high throughput）来访问应用程序的数据，适合那些有着超大数据集的应用程序。HDFS 放宽了可移植操作系统接口（POSIX，Portable Operating System Interface）的要求，这样就可以实现以流的形式访问文件系统中的

数据。HDFS 原本是开源的 Apache 项目 Nutch 的基础结构，最后它成为了 Hadoop 的基础架构之一。

以下是 HDFS 的设计目标：

（1）检测和快速恢复硬件故障。硬件故障是常见的问题，整个 HDFS 系统由数百台或数千台存储着数据文件的服务器组成，而如此多的服务器意味着高故障率，因此，故障的检测和自动快速恢复是 HDFS 的一个核心目标。

（2）流式的数据访问。HDFS 使应用程序能流式地访问它们的数据集。HDFS 被设计成适合进行批量处理，而不是用户交互式的处理。所以它重视数据吞吐量，而不是数据访问的反应速度。

（3）简化一致性模型。大部分的 HDFS 程序操作文件时需要一次写入，多次读取。一个文件一旦经过创建、写入、关闭之后就不需要修改了，从而简化了数据一致性问题和高吞吐量的数据访问问题。

（4）通信协议。所有的通信协议都在 TCP/IP 协议之上。一个客户端和明确配置了端口的目录节点（NameNode）建立连接之后，它和目录节点（NameNode）的协议便是客户端协议（Client Protocal）。数据节点（DataNode）和目录节点（NameNode）之间则用数据节点协议（DataNode Protocal）。

- **HBase**

HBase 位于结构化存储层，是一个分布式的列存储数据库；该技术来源于 Google 的论文“Bigtable：一个结构化数据的分布式存储系统”。如同 Bigtable 利用了 Google 文件系统（Google File System）提供的分布式数据存储方式一样，HBase 在 Hadoop 之上提供了类似于 Bigtable 的能力。HBase 是 Hadoop 项目的子项目。HBase 不同于一般的关系数据库，其一，HBase 是一个适合于存储非结构化数据的数据库；其二，HBase 是基于列而不是基于行的模式。HBase 和 Bigtable 使用相同的数据模型。用户将数据存储在一个表里，一个数据行拥有一个可选择的键和任意数量的列。由于 HBase 表示疏松的，用户可以给行定义各种不同的列。HBase 主要用于需要随机访问、实时读写的大数据（BigData）。

- **MapReduce**

Mapreduce 是一种编程模型，用于大规模数据集（大于 1TB）的并行运算。“映射”（map）、“化简”（reduce）等概念和它们的主要思想都是从函数式编程语言中借来的。它使得编程人员在不了解分布式并行编程的情况下也能方便地将自己的程序运行在分布式系统上。MapReduce 在执行时先指定一个 map（映射）函数，把输入键值对映射成一组新的键值对，

经过一定的处理后交给 reduce，reduce 对相同 key 下的所有 value 进行处理后再输出键值对作为最终的结果。

- **Zookeeper**

Zookeeper 是一个分布式的、高可用性的协调服务，提供分布式锁之类的基本服务，用于构建分布式应用，为 HBase 提供了稳定服务和失败恢复机制。

- **Hive**

Hive 最早是由 Facebook 设计的，是一个建立在 Hadoop 基础之上的数据仓库，它提供了一些对存储在 Hadoop 文件中的数据集进行数据整理、特殊查询和分析的工具。Hive 提供的是一种结构化数据的机制，它支持类似于传统 RDBMS 中的 SQL 语言来帮助那些熟悉 SQL 的用户查询 Hadoop 中的数据，该查询语言称为 HiveQL。与此同时，那些传统的 MapReduce 编程人员也可以在 Mapper 或 Reducer 中通过 HiveQL 查询数据。Hive 编译器会把 HiveQL 编译成一组 MapReduce 任务，从而方便 MapReduce 编程人员进行 Hadoop 应用的开发。

- **Pig**

Pig 是一种数据流语言和运行环境，用以检索非常大的数据集，大大简化了 Hadoop 常见的工作任务。Pig 可以加载数据、表达转换数据以及存储最终结果。Pig 内置的操作使得半结构化数据变得有意义（如日志文件）。Pig 和 Hive 都为 HBase 提供了高层语言支持，使得在 HBase 上进行数据统计处理变的非常简单；但是，二者还是有所区别的。Hive 在 Hadoop 中扮演数据仓库的角色，允许使用类似于 SQL 语法进行数据查询。Hive 更适合于数据仓库的任务，主要用于静态的结构以及需要经常分析的工作。Hive 与 SQL 相似，这一点使其成为 Hadoop 与其他 BI 工具结合的理想交集。Pig 赋予开发人员在大数据集领域更多的灵活性，并允许开发简洁的脚本用于转换数据流以便嵌入到较大的应用程序。与 Hive 相比较，Pig 属于较轻量级，它主要的优势是，相比于直接使用 Hadoop Java APIs 而言，使用 Pig 可以大幅削减代码量。

- **Sqoop**

Sqoop 为 HBase 提供了方便的 RDBMS 数据导入功能，使得传统数据库数据向 HBase 中迁移变得非常方便。

3.8 Hadoop 的体系结构

HDFS 和 MapReduce 是 Hadoop 的两大核心。而整个 Hadoop 的体系结构主要是通过

HDFS 来实现对分布式存储的底层支持的，并且它会通过 MapReduce 来实现对分布式并行任务处理的程序支持。

3.8.1 HDFS 的体系结构

我们首先介绍 HDFS 的体系结构。HDFS 采用了主从（Master/Slave）结构模型，一个 HDFS 集群是由一个 NameNode 和若干个 DataNode 组成的。其中 NameNode 作为主服务器，管理文件系统的命名空间和客户端对文件的访问操作；集群中的 DataNode 管理存储的数据。HDFS 允许用户以文件的形式存储数据。从内部来看，文件被分成若干个数据块，而且这若干个数据块存放在一组 DataNode 上。NameNode 执行文件系统的命名空间操作，比如打开、关闭、重命名文件或目录等，它也负责数据块到具体 DataNode 的映射。DataNode 负责处理文件系统客户端的文件读写请求，并在 NameNode 的统一调度下进行数据块的创建、删除和复制工作。图 3-3 给出了 HDFS 的体系结构。

NameNode 和 DataNode 都被设计成可以在普通商用计算机上运行。这些计算机通常运行的是 GNU/Linux 操作系统。HDFS 采用 Java 语言开发，因此，任何支持 Java 的机器都可以部署 NameNode 和 DataNode。一个典型的部署场景是集群中的一台机器运行一个 NameNode 实例，其他机器分别运行一个 DataNode 实例。当然，并不排除一台机器运行多个 DataNode 实例的情况。集群中单一的 NameNode 的设计则大大简化了系统的架构。NameNode 是所有 HDFS 元数据的管理者，用户数据永远不会经过 NameNode。

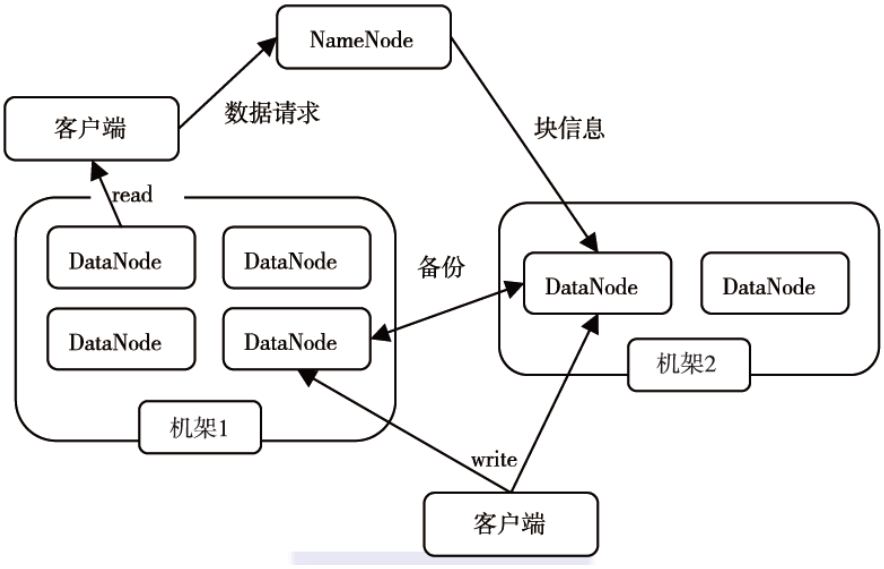

图 3-3 HDFS 的体系结构图

3.8.2 MapReduce 的体系结构

接下来介绍 MapReduce 的体系结构。MapReduce 是一种并行编程模式，这种模式使得软件开发者可以轻松地编写出分布式并行程序。在 Hadoop 的体系结构中，MapReduce 是一个简单易用的软件框架，基于它可以将任务分发到由上千台商用机器组成的集群上，并以一种高容错的方式并行处理大量的数据集，实现 Hadoop 的并行任务处理功能。MapReduce 框架是由一个单独运行在“主节点”上的 JobTracker 和运行在每个集群“从节点”上的 TaskTracker 共同组成的。主节点负责调度构成一个作业的所有任务，这些任务分布在不同的从节点上。主节点监控它们的执行情况，并且重新执行之前失败的任务；从节点仅负责由主节点指派的任务。当一个 Job 被提交时，JobTracker 接收到提交作业和配置信息之后，就会将配置信息等分发给从节点，同时调度任务并监控 TaskTracker 的执行。

从上面的介绍可以看出，HDFS 和 MapReduce 共同组成了 Hadoop 分布式系统体系结构的核心。HDFS 在集群上实现了分布式文件系统，MapReduce 在集群上实现了分布式计算和任务处理。HDFS 在 MapReduce 任务处理过程中提供了文件操作和存储等支持，MapReduce 在 HDFS 的基础上实现了任务的分发、跟踪、执行等工作，并收集结果，二者相互作用，完成了 Hadoop 分布式集群的主要任务。

3.9 Hadoop 与分布式开发

我们通常说的分布式系统其实是分布式软件系统，即支持分布式处理的软件系统，它是在通信网络互联的多处理机体系结构上执行任务的，包括分布式操作系统、分布式程序设计语言及其编译（解释）系统、分布式文件系统和分布式数据库系统等。Hadoop 是分布式软件系统中文件系统这一层的软件，它实现了分布式文件系统和部分分布式数据库的功能。

Hadoop 中的分布式文件系统 HDFS，能够实现数据在计算机集群组成的云上高效的存储和管理。Hadoop 中的并行编程框架 MapReduce，能够让用户编写的 Hadoop 并行应用程序运行更加简化。

Hadoop 上的并行应用程序开发是基于 MapReduce 编程框架的。MapReduce 编程模型的原理是：利用一个输入的 key/value 对集合来产生一个输出的 key/value 对集合。MapReduce 库的用户用两个函数来表达这个计算：Map 和 Reduce。

用户自定义的 map 函数接收一个输入的 key/value 对，然后产生一个中间 key/value 对的

集合。MapReduce 把所有具有相同 key 值的 value 集合在一起，然后传递给 reduce 函数。用户自定义的 reduce 函数接收 key 和相关的 value 集合。reduce 函数合并这些 value 值，形成一个较小的 value 集合。一般来说，每次 reduce 函数调用只产生 0 或 1 个输出的 value 值。通常我们通过一个迭代器把中间的 value 值提供给 reduce 函数，这样就可以处理无法全部放入内存中的大量的 value 值集合了。

图 3-4 是 MapReduce 的数据流图，这个过程简而言之就是将大数据集分解为成百上千个小数据集，每个（或若干个）数据集分别由集群中的一个节点（一般就是一台普通的计算机）进行处理并生成中间结果，然后这些中间结果又由大量的节点合并，形成最终结果。图 3-4 也指出了 MapReduce 框架下并行程序中的两个主要函数：map 和 reduce。在这个结构中，需要用户完成的工作仅仅是根据任务编写 map 和 reduce 两个函数。

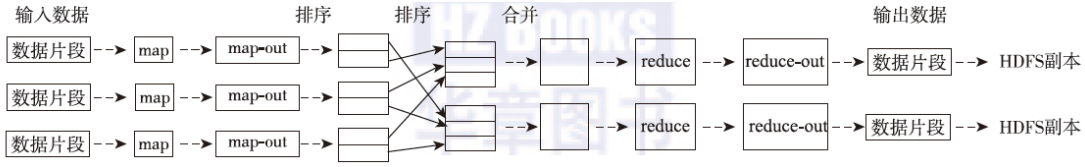

图 3-4MapReduce 的数据流图

MapReduce 计算模型非常适合在大量计算机组成的大规模集群上并行运行。图 3-4 中的每一个 map 任务和每一个 reduce 任务均可以同时运行于一个单独的计算节点上，可想而知，其运算效率是很高的，那么这样的并行计算是如何做到的呢？下面将简单介绍一下其原理。

1. 数据分布存储

Hadoop 分布式文件系统（HDFS）由一个目录节点（NameNode）和 N 个数据节点（DataNode）组成，每个节点均是一台普通的计算机。在使用方式上 HDFS 与我们熟悉的单机文件系统非常类似，它可以创建目录，创建、复制和删除文件，以及查看文件的内容等。但 HDFS 底层把文件切割成了 Block，然后这些 Block 分散地存储于不同的 DataNode 上，每个 Block 还可以复制数份数据存储于不同的 DataNode 上，达到容错容灾的目的。NameNode 则是整个 HDFS 的核心，它通过维护一些数据结构来记录每一个文件被切割成了多少个 Block、这些 Block 可以从哪些 DataNode 中获得，以及各个 DataNode 的状态等重要信息。

2. 分布式并行计算

Hadoop 中有一个作为主控的 JobTracker，用于调度和管理其他的 TaskTracker，JobTracker 可以运行于集群中的任意一台计算机上。TaskTracker 则负责执行任务，它必须运行于 DataNode 上，也就是说 DataNode 既是数据存储节点，也是计算节点。JobTracker 将 map 任务和 reduce 任务分发给空闲的 TaskTracker，让这些任务并行运行，并负责监控任务的运行

情况。如果某一个 TaskTracker 出了故障，JobTracker 会将其负责的任务转交给另一个空闲的 TaskTracker 重新运行。

3. 本地计算

数据存储在哪一台计算机上，就由哪台计算机进行这部分数据的计算，这样可以减少数据在网络上的传输，降低对网络带宽的需求。在 Hadoop 这类基于集群的分布式并行系统中，计算节点可以很方便地扩充，它所能够提供的计算能力近乎无限，但是由于数据需要在不同的计算机之间流动，故网络带宽变成了瓶颈，“本地计算”是一种最有效的节约网络带宽的手段，业界把这形容为“移动计算比移动数据更经济”。

4. 任务粒度

把原始大数据集切割成小数据集时，通常让小数据集小于或等于 HDFS 中一个 Block 的大小（默认是 64MB），这样能够保证一个小数据集是位于一台计算机上的，便于本地计算。有 M 个小数据集待处理，就启动 M 个 map 任务，注意这 M 个 map 任务分布于 N 台计算机上，它们会并行运行，reduce 任务的数量 R 则可由用户指定。

5. 数据分割（Partition）

把 map 任务输出的中间结果按 key 的范围划分成 R 份（R 是预先定义的 reduce 任务的个数），划分时通常使用 hash 函数（如：hash(key) mod R），这样可以保证某一范围内的 key 一定是由一个 reduce 任务来处理的，可以简化 Reduce 的过程。

6. 数据合并（Combine）

在数据分割之前，还可以先对中间结果进行数据合并（Combine），即将中间结果中有相同 key 的 <key, value> 对合并成一对。Combine 的过程与 reduce 的过程类似，很多情况下可以直接使用 reduce 函数，但 Combine 是作为 map 任务的一部分，在执行完 map 函数后紧接着执行的。Combine 能够减少中间结果中<key, value>对的数目，从而降低网络流量。

7. Reduce

Map 任务的中间结果在做完 Combine 和 Partition 之后，以文件形式存于本地磁盘上。中间结果文件的位置会通知主控 JobTracker，JobTracker 再通知 reduce 任务到哪一个 DataNode 上去取中间结果。注意，所有的 map 任务产生的中间结果均按其 key 值用同一个 hash 函数划分成了 R 份，R 个 reduce 任务各自负责一段 key 区间。每个 reduce 需要向许多个 map 任务节点取得落在其负责的 key 区间内的中间结果，然后执行 reduce 函数，形成一个最终的结果文件。

8. 任务管道

有 R 个 reduce 任务，就会有 R 个最终结果，很多情况下这 R 个最终结果并不需要合并成一个最终结果，因为这 R 个最终结果又可以作为另一个计算任务的输入，开始另一个并行计算任务，这也就形成了任务管道。

3.10Hadoop 应用案例

随着企业的数据量的迅速增长，存储和处理大规模数据已成为企业的迫切需求。Hadoop 作为开源的云计算平台，已引起了学术界和企业的普遍兴趣。

在学术方面，Hadoop 得到了各科研院所的广泛关注，多所著名大学加入到 Hadoop 集群的研究中来，其中包括斯坦福大学、加州大学伯克利分校、康奈尔大学、卡耐基.梅隆大学、普渡大学等。一些国内高校和科研院所如中科院计算所、清华大学、中国人民大学等也开始对 Hadoop 展开相关研究，研究内容涉及 Hadoop 的数据存储、资源管理、作业调度、性能优化、系统可用性和安全性等多个方面。

在商业方面，Hadoop 技术已经在互联网领域得到了广泛的应用。互联网公司往往需要存储海量的数据并对其进行处理，而这正是 Hadoop 的强项。如 Facebook 使用 Hadoop 存储内部的日志拷贝以及数据挖掘和日志统计；Yahoo！利用 Hadoop 支持广告系统并处理网页搜索；Twitter 则使用 Hadoop 存储微博数据、日志文件和其他中间数据等。在国内，Hadoop 同样也得到了许多公司的青睐，如百度主要将 Hadoop 应用于日志分析和网页数据库的数据挖掘；阿里巴巴则将 Hadoop 用于商业数据的排序和搜索引擎的优化等。

本章小结

本章首先介绍了 Hadoop 分布式计算平台：它是由 Apache 软件基金会开发的一个开源分布式计算平台。以 Hadoop 分布式文件系统（HDFS）和 MapReduce 为核心的 Hadoop 为用户提供了系统底层细节透明的分布式基础架构。由于 Hadoop 拥有可计量、成本低、高效、可信等突出特点，基于 Hadoop 的应用已经遍地开花，尤其是在互联网领域。

本章接下来介绍了 Hadoop 项目及其结构，现在 Hadoop 已经发展成为一个包含多个子项目的集合，被用于分布式计算，虽然 Hadoop 的核心是 Hadoop 分布式文件系统和 MapReduce，但 Hadoop 下的 Avro、Hive、HBase 等子项目提供了互补性服务或在核心层之上提供了更高层的服务；接下来简要介绍了以 HDFS 和 MapReduce 为核心的 Hadoop 体系结构。

最后，介绍了 Hadoop 的典型应用案例。

参考文献

[1] 陆嘉恒. Hadoop 实战. 机械工业出版社. 2011 年.

[2] 曾大聃，周傲英(译). Hadoop 权威指南中文版. 清华大学出版社. 2010 年.

第 4 章 MapReduce

MapReduce 是一种编程模型，用于大规模数据集（大于 1TB）的并行运算。概念“Map（映射）”和“Reduce（归约）”以及它们的主要思想，都是从函数式编程语言里借用而来的，同时也包含了从矢量编程语言里借来的特性。MapReduce 极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。

本章介绍 MapReduce 的相关知识，内容要点如下：

- 分布式并行编程：编程方式的变革
- MapReduce 模型概述
- Map 和 Reduce 函数
- MapReduce 工作流程
- 并行计算的实现
- 实例分析：WordCount
- 新 MapReduce 框架 Yarn

4.1 分布式并行编程：编程方式的变革

根据摩尔定律，约每隔 18 个月，CPU 性能会提高一倍。然而，由于晶体管电路已经逐渐接近其物理上的性能极限，摩尔定律在 2005 年左右开始失效。随着互联网时代的到来，软件编程方式发生了重大的变革，基于大规模计算机集群的分布式并行编程是将来软件性能提升的主要途径。基于集群的分布式并行编程能够让软件与数据同时运行在连成一个网络的许多台计算机上，由此获得海量计算能力。

在摩尔定律的作用下，以前程序员根本不用考虑计算机的性能会跟不上软件的发展，因为约每隔 18 个月，CPU 的主频就会增加一倍，性能也将提升一倍，软件根本不用做任何改变，就可以享受免费的性能提升。然而，由于晶体管电路已经逐渐接近其物理上的性能极限，摩尔定律在 2005 年左右开始失效了，人类再也不能期待单个 CPU 的速度每隔 18 个月就翻一倍，为我们提供越来越快的计算性能。Intel、AMD、IBM 等芯片厂商开始从多核这个角度来挖掘 CPU 的性能潜力，多核时代以及互联网时代的到来，将使软件编程方式发生重大变革，基于多核的多线程并发编程以及基于大规模计算机集群的分布式并行编程是将来软件

性能提升的主要途径。

许多人认为这种编程方式的重大变化将带来一次软件的并发危机，因为我们传统的软件方式基本上是单指令单数据流的顺序执行，这种顺序执行十分符合人类的思考习惯，却与并发并行编程格格不入。基于集群的分布式并行编程，能够让软件与数据同时运行在连成一个网络的许多台计算机上，这里的每一台计算机均可以是一台普通的 PC 机。这样的分布式并行环境的最大优点是，可以很容易地通过增加计算机来扩充新的计算节点，并由此获得不可思议的海量计算能力，同时又具有相当强的容错能力，一批计算节点失效也不会影响计算的正常进行以及结果的正确性。Google 就是这么做的，他们使用了叫做 MapReduce 的并行编程模型进行分布式并行编程，运行在叫做 GFS (Google File System)的分布式文件系统上，为全球亿万用户提供搜索服务。

Hadoop 实现了 Google 的 MapReduce 编程模型，提供了简单易用的编程接口，也提供了它自己的分布式文件系统 HDFS，与 Google 不同的是，Hadoop 是开源的，任何人都可以使用这个框架来进行并行编程。如果说分布式并行编程的难度足以让普通程序员望而生畏的话，开源的 Hadoop 的出现，则极大地降低了它的门槛。你会发现，基于 Hadoop 编程非常简单，无需任何并行开发经验，你也可以轻松地开发出分布式的并行程序，并让其令人难以置信地同时运行在数百台机器上，然后在短时间内完成海量数据的计算。你可能会觉得你不可能会拥有数百台机器来运行你的并行程序，而事实上，随着“云计算”的普及，任何人都可以轻松获得这样的海量计算能力。例如，现在 Amazon 公司的云计算平台 Amazon EC2 已经提供了这种按需计算的租用服务。

掌握一点分布式并行编程的知识对将来的程序员是必不可少的，Hadoop 是如此地简便好用，何不尝试一下呢？也许你已经急不可耐地想试一下基于 Hadoop 的编程是怎么回事了，但毕竟这种编程模型与传统的顺序程序大不相同，掌握一点基础知识才能更好地理解基于 Hadoop 的分布式并行程序是如何编写和运行的。因此，这里会先介绍一下 MapReduce 的计算模型和 Hadoop 中的分布式文件系统 HDFS，然后介绍 Hadoop 是如何实现并行计算的。

4.2 MapReduce 模型概述

MapReduce 是 Google 公司的核心计算模型，这是一个令人惊讶的、简单却又威力巨大的模型，它将复杂的、运行于大规模集群上的并行计算过程高度地抽象到了两个函数：Map 和 Reduce。适合用 MapReduce 来处理的数据集(或任务)，需要满足一个基本要求：待处理的

数据集可以分解成许多小的数据集，而且每一个小数据集都可以完全并行地进行处理。概念“Map（映射）”和“Reduce（归约）”，以及它们的主要思想，都是从函数式编程语言里借来的，同时包含了从矢量编程语言里借来的特性。MapReduce 极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。

一个 MapReduce 作业（job）通常会把输入的数据集切分为若干独立的数据块，由 map 任务（task）以完全并行的方式处理它们。框架会对 map 的输出先进行排序，然后把结果输入给 reduce 任务。通常作业的输入和输出都会被存储在文件系统中。整个框架负责任务的调度和监控，以及重新执行已经失败的任务。

通常，MapReduce 框架和分布式文件系统是运行在一组相同的节点上的，也就是说，计算节点和存储节点通常在一起。这种配置允许框架在那些已经存好数据的节点上高效地调度任务，这可以使整个集群的网络带宽被非常高效地利用。

MapReduce 框架由单独一个 master JobTracker 和每个集群节点一个 slave TaskTracker 共同组成。这个 master 负责调度构成一个作业的所有任务，这些任务分布在不同的 slave 上，master 监控它们的执行，重新执行已经失败的任务。而 slave 仅负责执行由 master 指派的任务。

应用程序至少应该指明输入/输出的位置（路径），并通过实现合适的接口或抽象类提供 map 和 reduce 函数，再加上其他作业的参数，就构成了作业配置（job configuration）。然后，Hadoop 的 job client 提交作业（jar 包/可执行程序等）和配置信息给 JobTracker，后者负责分发这些软件和配置信息给 slave、调度任务且监控它们的执行，同时提供状态和诊断信息给 job client。

虽然 Hadoop 框架是用 Java 实现的，但 MapReduce 应用程序则不一定要用 Java 来写。

4.3 Map 和 Reduce 函数

MapReduce 计算模型的核心是 map 和 reduce 两个函数，这两个函数由用户负责实现，功能是按一定的映射规则将输入的<key，value>转换成另一个或一批<key，value>输出（见表 4-1）。

表 4-1 Map 和 Reduce

函数	输入	输出	说明
Map	<k1, v1>	List(<k2,v2>)	1. 将小数据集进一步解析成一批 <key,value> 对，输入 Map 函数中进行处理。 2. 每一个输入的 <k1,v1> 会输出一批 <k2,v2>。 <k2,v2> 是计算的中间结果。
Reduce	<k2,List(v2)>	<k3,v3>	输入的中间结果 <k2,List(v2)> 中的 List(v2) 表示是一批属于同一个 k2 的 value

以一个计算文本文件中每个单词出现的次数的程序为例，<k1，v1>可以是 <行在文件中的偏移位置，文件中的一行>，经 Map 函数映射之后，形成一批中间结果 <单词，出现次数>，而 Reduce 函数则可以对中间结果进行处理，将相同单词的出现次数进行累加，得到每个单词的总的出现次数。

基于 MapReduce 计算模型编写分布式并行程序非常简单，程序员的主要编码工作就是实现 Map 和 Reduce 函数，其它的并行编程中的种种复杂问题，如分布式存储、工作调度、负载均衡、容错处理、网络通信等，均由 MapReduce 框架(比如 Hadoop)负责处理，程序员完全不用操心。

4.4 MapReduce 工作流程

4.4.1 工作流程概述

图 4-1 说明了用 MapReduce 来处理大数据集的过程，这个 MapReduce 的计算过程简而言之，就是将大数据集分解为成百上千的小数据集，每个(或若干个)数据集分别由集群中的一个节点(一般就是一台普通的计算机)进行处理并生成中间结果，然后这些中间结果又由大量的节点进行合并，形成最终结果。

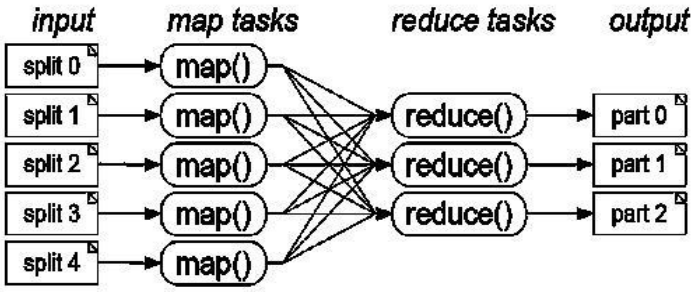

图 4-1 MapReduce 工作流程

MapReduce 的输入一般来自 HDFS 中的文件，这些文件分布存储在集群内的节点上。运行一个 MapReduce 程序会在集群的许多节点甚至所有节点上运行 mapping 任务，每一个 mapping 任务都是平等的：mappers 没有特定“标识物”与其关联。因此，任意的 mapper

都可以处理任意的输入文件。每一个 mapper 会加载一些存储在运行节点本地的文件集来进行处理（注：这是移动计算，把计算移动到数据所在节点，可以避免额外的数据传输开销）。

当 mapping 阶段完成后，这阶段所生成的中间“键值对”数据必须在节点间进行交换，把具有相同键的数值发送到同一个 reducer 那里。Reduce 任务在集群内的分布节点同 mappers 的一样。这是 MapReduce 中唯一的任务节点间的通信过程。map 任务之间不会进行任何的信息交换，也不会去关心别的 map 任务的存在。相似地，不同的 reduce 任务之间也不会有通信。用户不能显式地从一台机器发送信息到另外一台机器；所有数据传送都是由 Hadoop MapReduce 平台自身去做的，这些是通过关联到数值上的不同键来隐式引导的。这是 Hadoop MapReduce 的可靠性的基础元素。如果集群中的节点失效了，任务必须可以被重新启动。如果任务已经执行了有副作用（side-effect）的操作，比如说，跟外面进行通信，那共享状态必须存在可以重启的任务上。消除了通信和副作用问题，那重启就可以做得更优雅些。

4.4.2 MapReduce 各个执行阶段

一般而言，Hadoop 的一个简单的 MapReduce 任务执行流程如下（如图 4-2 所示）：

1) JobTracker 负责分布式环境中实现客户端创建任务并提交；

2)InputFormat 模块负责做 Map 前的预处理，主要包括以下几个工作：验证输入的格式是否符合 JobConfig 的输入定义，可以是专门定义或者是 Writable 的子类。将 input 的文件切分为逻辑上的输入 InputSplit，因为在分布式文件系统中 blocksize 是有大小限制的，因此大文件会被划分为多个较小的 block。通过 RecordReader 来处理经过文件切分为 InputSplit 的一组 records，输出给 Map。因为 InputSplit 是逻辑切分的第一步，如何根据文件中的信息来具体切分还需要 RecordReader 完成。

3)将 RecordReader 处理后的结果作为 Map 的输入，然后 Map 执行定义的 Map 逻辑，输出处理后的<key,value>对到临时中间文件。

4)Shuffle&Partitioner：在 MapReduce 流程中，为了让 reduce 可以并行处理 map 结果，必须对 map 的输出进行一定的排序和分割，然后再交给对应的 reduce，而这个将 map 输出进行进一步整理并交给 reduce 的过程，就称为 shuffle。Partitioner 是选择配置，主要作用是在多个 Reduce 的情况下，指定 Map 的结果由某一个 Reduce 处理，每一个 Reduce 都会有单独的输出文件。

6)Reduce 执行具体的业务逻辑，即用户编写的处理数据得到结果的业务，并且将处理

结果输出给 OutputFormat。

7)OutputFormat 的作用是，验证输出目录是否已经存在以及输出结果类型是否符合 Config 中配置类型，如果都成立，则输出 Reduce 汇总后的结果。

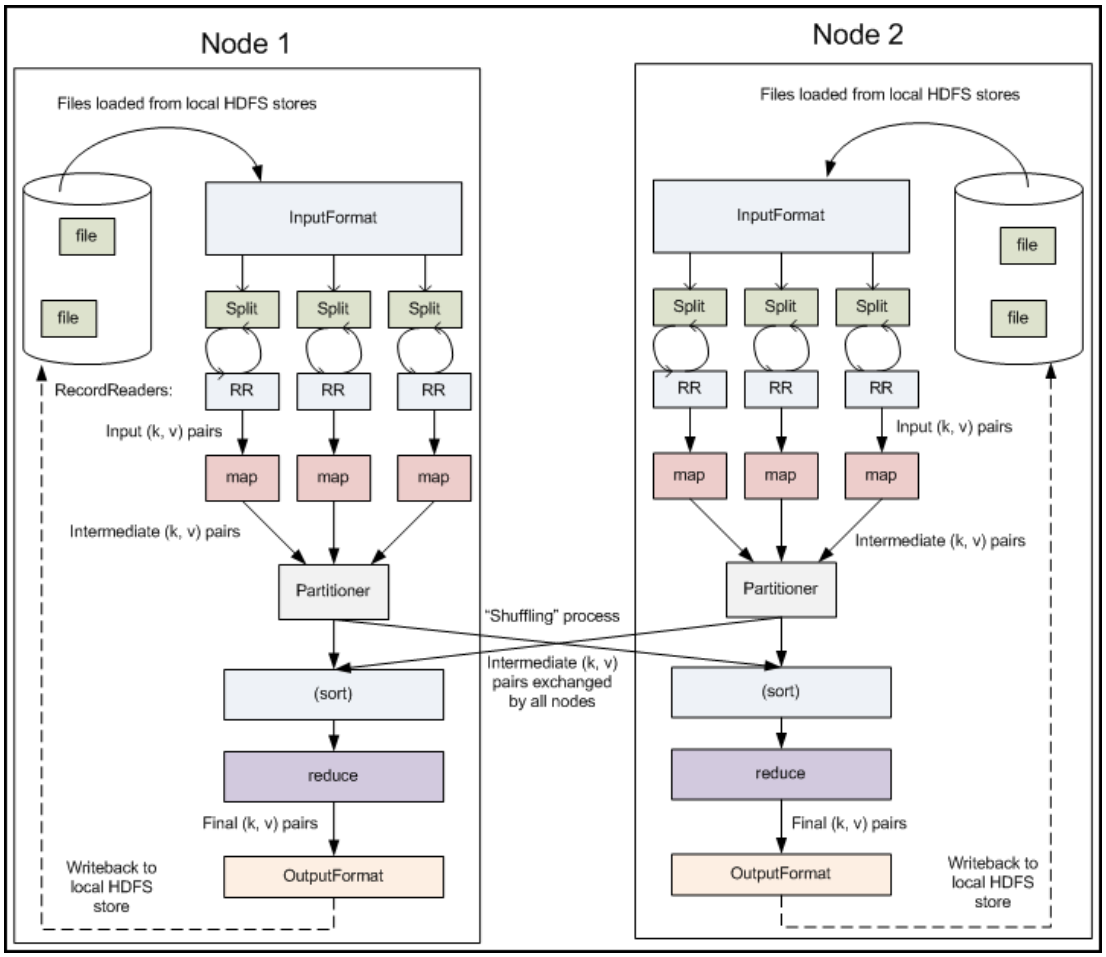

图 4-2 Hadoop MapReduce 工作流程中的各个执行阶段

下面简单介绍一下 MapReduce 过程的各个部分，包括输入文件、输入格式、输入块、记录读取器、Mapper、Partion&Shuffle、Reducer、输出格式、RecordWriter。

- **输入文件**

文件是 MapReduce 任务的数据的初始存储地。正常情况下，输入文件一般是存在 HDFS 里。这些文件的格式可以是任意的；我们可以使用基于行的日志文件，也可以使用二进制格式，多行输入记录或其它一些格式。这些文件会很大——数十 GB 或更大。

- **输入格式**

InputFormat 类定义了如何分割和读取输入文件，它提供了以下几个功能：

（1） 选择作为输入的文件或对象；

（2） 定义把文件划分到任务的 InputSplits；

（3） 为 RecordReader 读取文件提供了一个工厂方法。

Hadoop 自带了好几个输入格式。其中有一个抽象类叫 FileInputFormat，所有操作文件的 InputFormat 类都是从它那里继承功能和属性。当开启 Hadoop 作业时，FileInputFormat 会得到一个路径参数，这个路径内包含了所需要处理的文件，FileInputFormat 会读取这个文件夹内的所有文件（注：默认不包括子文件夹内的），然后它会把这些文件拆分成一个或多个 InputSplit。你可以通过 JobConf 对象的 setInputFormat()方法来设定应用到你的作业输入文件上的输入格式。表 4-2 给出了 MapReduce 提供的输入格式。

表 4-2 MapReduce 提供的输入格式

输入格式	描述	键	值
TextInputFormat	默认格式，读取文件的行	行的字节偏移量	行的内容
KeyValueInputFormat	把行解析为键值对	第一个 tab 字符前的所有字符	行剩下的内容
SequenceFileInputFormat	Hadoop 定义的高性能二进制格式	用户自定义	用户自定义

默认的输入格式是 TextInputFormat，它把输入文件每一行作为单独的一个记录，但不做解析处理。这对那些没有被格式化的数据或是基于行的记录来说是很有用的，比如日志文件。更有趣的一个输入格式是 KeyValueInputFormat，这个格式也是把输入文件每一行作为单独的一个记录。然而不同的是，TextInputFormat 把整个文件行当做值数据，KeyValueInputFormat 则是通过搜寻 tab 字符来把行拆分为“键值对”。这在把一个 MapReduce 的作业输出作为下一个作业的输入时显得特别有用，因为，默认的 map 输出格式正是按 KeyValueInputFormat 格式输出数据。最后来讲讲 SequenceFileInputFormat，它用于读取特殊的、特定于 Hadoop 的二进制文件，这些文件包含了很多能让 Hadoop 的 mapper 快速读取数据的特性。Sequence 文件是块压缩的，并提供了对几种数据类型（不仅仅是文本类型）直接的序列化与反序列化操作。Squence 文件可以作为 MapReduce 任务的输出数据，并且用它做一个 MapReduce 作业到另一个作业的中间数据是很高效的。

- **输入块（InputSplit）**

一个输入块描述了构成 MapReduce 程序中单个 map 任务的一个单元。把一个 MapReduce 程序应用到一个数据集上，即是指一个作业，会由几个（也可能几百个）任务组成。Map 任务可能会读取整个文件，但一般是读取文件的一部分。默认情况下，FileInputFormat 及其

子类会以 64MB 为基数来拆分文件（与 HDFS 的 Block 默认大小相同，Hadoop 建议 Split 大小与此相同）。你可以在 hadoop-site.xml（注：0.20.*以后是在 mapred-default.xml 里）文件内设定 mapred.min.split.size 参数来控制具体划分大小，或者在具体 MapReduce 作业的 JobConf 对象中重写这个参数。通过以块形式处理文件，我们可以让多个 map 任务并行地操作一个文件。如果文件非常大的话，这个特性可以通过并行处理大幅地提升性能。更重要的是，因为多个块（Block）组成的文件可能会分散在集群内的好几个节点上，这样就可以把任务调度在不同的节点上；因此，所有的单个块都是本地处理的，而不是把数据从一个节点传输到另外一个节点。当然，日志文件可以以明智的块处理方式进行处理，但是，有些文件格式不支持块处理方式。针对这种情况，你可以写一个自定义的 InputFormat，这样你就可以控制你文件是如何被拆分（或不拆分）成文件块的。

输入格式定义了组成 mapping 阶段的 map 任务列表，每一个任务对应一个输入块。接下来根据输入文件块所在的物理地址，这些任务会被分派到对应的系统节点上，可能会有多个 map 任务被分派到同一个节点上。任务分派好后，节点开始运行任务，尝试去最大并行化执行。节点上的最大任务并行数由 mapred.tasktracker.map.tasks.maximum 参数控制。

- **记录读取器（RecordReader）**

InputSplit 定义了如何切分工作，但是没有描述如何去访问它。RecordReader 类则是实际地用来加载数据，并把数据转换为适合 mapper 读取的键值对。RecordReader 实例是由输入格式定义的，默认的输入格式 TextInputFormat，提供了一个 LineRecordReader，这个类会把输入文件的每一行作为一个新的值，关联到每一行的键则是该行在文件中的字节偏移量。RecordReader 会在输入块上被重复地调用，直到整个输入块被处理完毕，每一次调用 RecordReader 都会调用 Mapper 的 map()方法。

- **Mapper**

Mapper 执行了 MapReduce 程序第一阶段中有趣的用户定义的工作。给定一个键值对，map()方法会生成一个或多个键值对，这些键值对会被送到 Reducer 那里。对于整个作业输入部分的每一个 map 任务（输入块），每一个新的 Mapper 实例都会在单独的 Java 进程中被初始化，mapper 之间不能进行通信。这就使得每一个 map 任务的可靠性不受其它 map 任务的影响，只由本地机器的可靠性来决定。

- **Partition & Shuffle**

当第一个 map 任务完成后，节点可能还要继续执行更多的 map 任务，但这时候也开始把 map 任务的中间输出交换到需要它们的 reducer 那里去，这个把“map 输出”移动到 reducer

那里去的过程就叫做 shuffle。每一个 reduce 节点会被分配得到“map 输出的键集合”中的一个不同的子集合，这些子集合（被称为“partitions”）是 reduce 任务的输入数据。每一个 map 任务生成的键值对，可能会隶属于任意的 partition，有着相同键的数值总是在一起被 reduce，不管它是来自那个 mapper 的。因此，所有的 map 节点必须就把不同的中间数据发往何处达成一致。Partitioner 类就是用来决定给定键值对的去向，默认的分类器（partitioner）会计算键的哈希值，并基于这个结果来把键赋到相应的 partition 上。

- **Reducer**

每一个 reduce 任务负责对那些关联到相同键上的所有数值进行归约（reducing），每一个节点收到的中间键集合，在被送到具体的 reducer 那里前就已经自动被 Hadoop 排序过了。每个 reduce 任务都会创建一个 Reducer 实例，这是一个用户自定义代码的实例，负责执行特定作业的第二个重要的阶段。对于每一个已被赋予到 reducer 的 partition 内的键来说，reducer 的 reduce()方法只会调用一次，它会接收一个键和关联到键的所有值的一个迭代器，迭代器会以一个未定义的顺序返回关联到同一个键的值。reducer 也要接收一个 OutputCollector 和 Report 对象，它们像在 map()方法中那样被使用。

- **输出格式**

提供给 OutputCollector 的键值对会被写到输出文件中，写入的方式由输出格式控制。OutputFormat 的功能跟前面描述的 InputFormat 类很像，Hadoop 提供的 OutputFormat 的实例会把文件写在本地磁盘或 HDFS 上，它们都是继承自公共的 FileInputFormat 类。每一个 reducer 会把结果输出写在公共文件夹中一个单独的文件内，这些文件的命名一般是 part-nnnnn，其中，nnnnn 是关联到某个 reduce 任务的 partition 的 id，输出文件夹通过 FileOutputFormat.setOutputPath()来设置。你可以通过具体 MapReduce 作业的 JobConf 对象的 setOutputFormat()方法来设置具体用到的输出格式。表 4-3 给出了 Hadoop 已提供的输出格式。

表 4-3 Hadoop 提供的输出格式

输出格式	描述
TextOutputFormat	默认的输出格式， 以 "key \t value" 的方式输出行
SequenceFileOutputFormat	输出二进制文件，适合于读取为子 MapReduce 作业的输入
NullOutputFormat	忽略收到的数据，即不做输出

Hadoop 提供了一些 OutputFormat 实例用于写入文件，基本的（默认的）实例是

TextOutputFormat，它会以一行一个键值对的方式把数据写入一个文本文件里。这样后面的 MapReduce 任务就可以通过 KeyValueInputFormat 类，简单地重新读取所需的输入数据了，而且也适合于人的阅读。还有一个更适合于在 MapReduce 作业间使用的中间格式，那就是 SequenceFileOutputFormat，它可以快速地序列化任意的数据类型到文件中，而对应 SequenceFileInputFormat 则会把文件反序列化为相同的类型并提交为下一个 Mapper 的输入数据，方式和前一个 Reducer 的生成方式一样。NullOutputFormat 不会生成输出文件并丢弃任何通过 OutputCollector 传递给它的键值对，如果你在要 reduce()方法中显式地写你自己的输出文件，并且不想 Hadoop 框架输出额外的空输出文件，那这个类是很有用的。

- **RecordWriter**

这个跟 InputFormat 中通过 RecordReader 读取单个记录的实现很相似，OutputFormat 类是 RecordWriter 对象的工厂方法，用来把单个的记录写到文件中，就像是 OuputFormat 直接写入的一样。

Reducer 输出的文件会留在 HDFS 上供你的其它应用使用，比如，另外一个 MapReduce 作业，或者一个给人工检查的单独程序。

4.4.3 Shuffle 过程详解

在 MapReduce 流程中，为了让 reduce 可以并行处理 map 结果，必须对 map 的输出进行一定的排序和分割，然后再交给对应的 reduce，而这个将 map 输出进行进一步整理并交给 reduce 的过程，就称为 shuffle。Shuffle 过程是 MapReduce 工作流程的核心，也被称为奇迹发生的地方。要想理解 MapReduce，Shuffle 是必须要了解的。

从图 4-3 中可以看出，shuffle 过程包含在 map 和 reduce 两端中，描述着数据从 map task 输出到 reduce task 输入的这段过程。在 map 端的 shuffle 过程是对 map 的结果进行划分（partition）、排序（sort）和 spill（溢写），然后，将属于同一个划分的输出合并在一起，并写到磁盘上，同时按照不同的划分将结果发送给对应的 reduce（map 输出的划分与 reduce 的对应关系由 JobTracker 确定）。reduce 端又会将各个 map 送来的属于同一个划分的输出进行合并（merge），然后对合并的结果进行排序，最后交给 reduce 处理。下面将从 map 和 reduce 两端详细介绍 shuffle 过程。

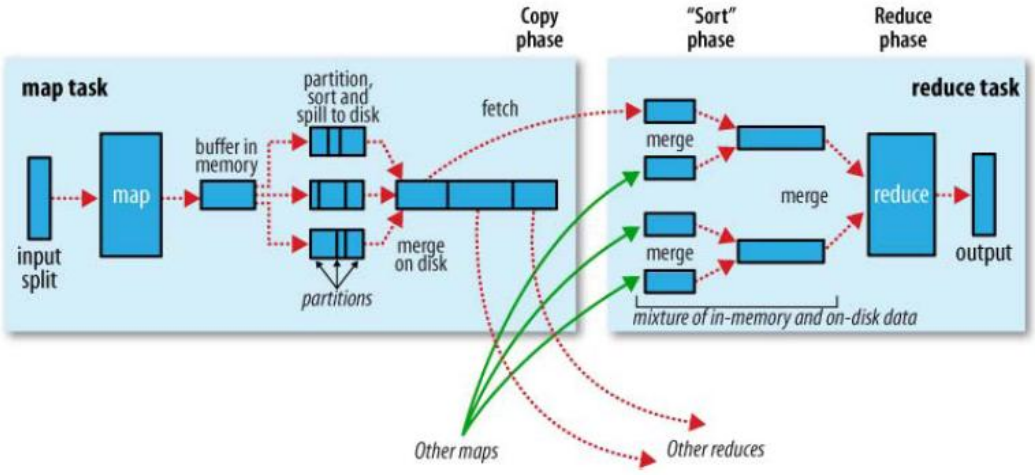

图 4-3 Shuffle 过程

4.4.3.1 map 端的 shuffle 过程

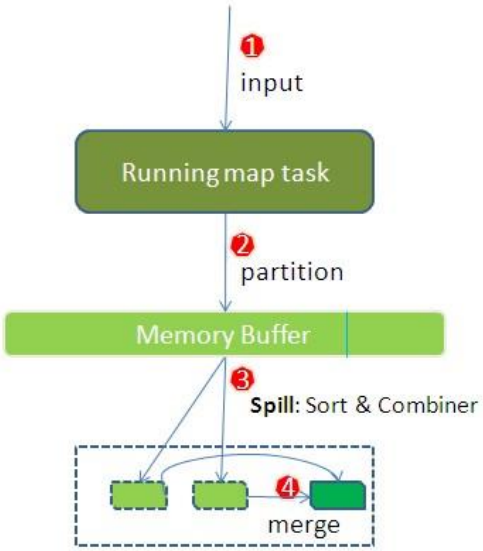

图 4-4 map 端的 shuffle 过程

图 4-4 是某个假想的 map task 的运行情况，可以清楚地说明划分（partition）、排序（sort）与合并（combiner）到底作用在 MapReduce 工作流程的哪个阶段，希望让大家清晰地了解从 map 数据输入到 map 端所有数据准备好的全过程。

整个流程可以包含四步。简单地说，每个 map task 都有一个内存缓冲区，存储着 map 的输出结果，当缓冲区快满的时候，需要将缓冲区的数据以一个临时文件的方式存放到磁盘，当整个 map task 结束后，再对磁盘中这个 map task 产生的所有临时文件做合并，生成最终的正式输出文件，然后等待 reduce task 来拉数据。当然这里的每一步都可能包含着多个步骤与细节，下面对每个步骤的细节进行说明。

第 1 步：在 map task 执行时，它的输入数据来源于 HDFS 的 block，当然在 MapReduce 概念中，map task 只读取 split。Split 与 block 的对应关系可能是多对一，默认是一对一。在 WordCount 例子里，假设 map 的输入数据都是像“aaa”这样的字符串。

第 2 步：在经过 mapper 的运行后，我们得知 mapper 的输出是这样一个 key/value 对：key 是“aaa”，value 是数值 1。注意，当前 map 端只做加 1 的操作，在 reduce task 里才去合并结果集。假设这个 job 有 3 个 reduce task，到底当前的“aaa”应该交由哪个 reduce 去做呢，是需要现在决定的。

MapReduce 提供 Partitioner 接口，它的作用就是根据 key 或 value 及 reducer 的数量来决定当前的这个“键值对”输出数据最终应该交由哪个 reduce task 处理。默认对 key 进行哈希以后再用 reduce task 数量进行取模，即 hash(key) mod R，其中 R 表示 reducer 的数量。默认的取模方式只是为了平均 reduce 的处理能力，如果用户自己对 Partitioner 有需求，可以订制并设置到 job 上。

假设在我们的例子中，“aaa”经过 Partitioner 后返回 0，也就是这个“键值对”应当交由第一个 reducer 来处理。接下来，需要将数据写入内存缓冲区中，缓冲区的作用是批量收集 map 结果，减少磁盘 IO 的影响。我们的 key/value 对以及 Partition 的结果都会被写入缓冲区。当然写入之前，key 与 value 值都会被序列化成字节数组。整个内存缓冲区就是一个字节数组。

第 3 步：这个内存缓冲区是有大小限制的，默认是 100MB。当 map task 的输出结果很多时，就可能会撑爆内存，所以，需要在一定条件下将缓冲区中的数据临时写入磁盘，然后重新利用这块缓冲区。这个从内存往磁盘写数据的过程被称为 Spill，中文可译为“溢写”，字面意思很直观。这个溢写是由单独线程来完成，不影响往缓冲区写 map 结果的线程。溢写线程启动时，不应该阻止 map 的结果输出，所以，整个缓冲区有个溢写的比例 spill.percent。这个比例默认是 0.8，也就是当缓冲区的数据已经达到阈值（buffer size * spill percent = 100MB * 0.8 = 80MB），溢写线程启动，锁定这 80MB 的内存，执行溢写过程。Map task 的输出结果还可以往剩下的 20MB 内存中写，互不影响。

当溢写线程启动后，需要对这 80MB 空间内的 key 做排序(Sort)。排序是 MapReduce 模型默认的行为，这里的排序也是对序列化的字节做的排序。

另外，为了进一步减少从 map 端到 reduce 端需要传输的数据量，还可以在 map 端执行 combine 操作。对于 WordCount 例子，就是简单地统计单词出现的次数，如果在同一个 map task 的结果中有很多个像“aaa”一样出现多次的 key，我们就应该把它们的值合并到一块，

这个过程叫 reduce，也叫 combine。比如，有些 map 输出数据可能像这样：<”aaa”, 1>，<”aaa”, 1>，经过 combine 操作以后就可以得到<”aaa”, 2>。但是，在 MapReduce 的术语中，reduce 单纯是指 reduce 端执行从多个 map task 取数据做计算的过程。除 reduce 外，非正式地合并数据只能算做 combine 了。实际上，MapReduce 中将 Combiner 等同于 Reducer。

如果 client 设置过 Combiner，那么现在就是使用 Combiner 的时候了。将有相同 key 的 key/value 对的 value 加起来，减少溢写到磁盘的数据量。Combiner 会优化 MapReduce 的中间结果，所以它在整个模型中会多次使用。那么，哪些场景才能使用 Combiner 呢？从这里分析，Combiner 的输出是 Reducer 的输入，Combiner 绝不能改变最终的计算结果。所以，一般而言，Combiner 只应该用于那种 Reduce 的输入 key/value 与输出 key/value 类型完全一致、且不影响最终结果的场景，比如累加、最大值等。Combiner 的使用一定得慎重，如果用好，它对 job 执行效率有帮助，反之，则会影响 reduce 的最终结果。

第 4 步：每次溢写会在磁盘上生成一个溢写文件，如果 map 的输出结果真的很大，有多次这样的溢写发生，磁盘上相应的就会有多个溢写文件存在。当 map task 真正完成时，内存缓冲区中的数据也全部溢写到磁盘中形成一个溢写文件。最终，磁盘中会至少有一个这样的溢写文件存在(如果 map 的输出结果很少，当 map 执行完成时，只会产生一个溢写文件)，因为最终的文件只允许有一个，所以需要将这些溢写文件归并到一起，这个过程就叫做 Merge。Merge 是怎样的？如前面的例子，“aaa”从某个 map task 读取过来时值是 5，从另外一个 map 读取时值是 8，因为它们有相同的 key，所以得 merge 成 group。什么是 group？对于“aaa”而言，merge 后得到的 group 就是像这样的：<”aaa”, {5, 8, 2, …}>，数组中的值就是从不同溢写文件中读取出来的，然后再把这些值合并起来。请注意，因为 merge 是将多个溢写文件合并到一个文件，所以可能也有相同的 key 存在，在这个过程中如果 client 设置过 Combiner，也会使用 Combiner 来合并相同的 key。

至此，map 端的所有工作都已结束，最终生成的这个文件也存放在 TaskTracker 够得着的某个本地目录内。每个 reduce task 不断地通过 RPC 从 JobTracker 那里获取 map task 是否完成的信息，如果 reduce task 得到通知，获知某台 TaskTracker 上的 map task 执行完成，Shuffle 的后半段过程开始启动。

4.4.3.2 reduce 端的 shuffle 过程

简单地说，reduce task 在执行之前的工作就是不断地拉取当前 job 里每个 map task 的最

终结果，然后对从不同地方拉取过来的数据不断地做 merge，也最终形成一个文件作为 reduce task 的输入文件（如图 4-5 所示）。

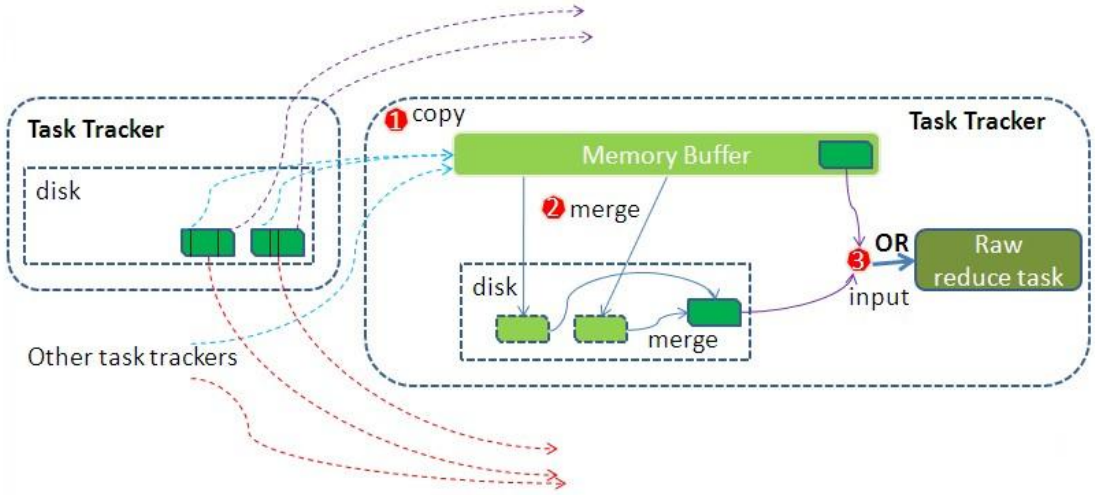

图 4-5 reduce 端的 shuffle 过程

和 map 端的细节图（图 4-4）一样，Shuffle 在 reduce 端的过程也能用图 4-5 上标明的 3 个步骤来概括。当前 reduce 拉取数据的前提是，它要从 JobTracker 那里获知有哪些 map task 已执行结束。Reducer 真正运行之前，所有的时间都是在拉取数据，做 merge，且不断重复地在做。如前面的方式一样，下面分成 3 步描述 reduce 端的 Shuffle 细节。

第 1 步：复制过程，简单地拉取数据。Reduce 进程启动一些数据复制线程(Fetcher)，通过 HTTP 方式请求 map task 所在的 TaskTracker 获取 map task 的输出文件。因为 map task 早已结束，这些文件就归 TaskTracker 管理在本地磁盘中。

第 2 步：Merge 阶段。这里的 merge 如 map 端的 merge 动作，只是数组中存放的是不同 map 端复制来的数值。复制过来的数据会先放入内存缓冲区中，这里的缓冲区大小要比 map 端的更为灵活，因为 Shuffle 阶段 Reducer 不运行，所以应该把绝大部分的内存都给 Shuffle 用。这里需要强调的是，merge 有三种形式：1)内存到内存；2)内存到磁盘；3)磁盘到磁盘。默认情况下第一种形式不启用，让人比较困惑，是吧。当内存中的数据量到达一定阈值，就启动内存到磁盘的 merge。与 map 端类似，这也是溢写的过程，这个过程中如果你设置有 Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种 merge 方式一直在运行，直到没有 map 端的数据时才结束，然后启动第三种磁盘到磁盘的 merge 方式生成最终的那个文件。

第 3 步：Reducer 的输入文件。不断地 merge 后，最后会生成一个“最终文件”。为什么加引号？因为这个文件可能存在于磁盘上，也可能存在于内存中。对我们来说，当然希望

它存放于内存中，直接作为 Reducer 的输入，但默认情况下，这个文件是存放于磁盘中的。当 Reducer 的输入文件已定，整个 Shuffle 才最终结束。然后就是 Reducer 执行，把结果放到 HDFS 上。

4.5 并行计算的实现

MapReduce 计算模型非常适合在大量计算机组成的大规模集群上并行运行。图 4-1 中的每一个 Map 任务和每一个 Reduce 任务均可以同时运行于一个单独的计算节点上，可想而知，其运算效率是很高的，那么这样的并行计算是如何做到的呢？实际上，这里涉及到了三个方面的内容（见图 4-6）：数据分布存储、分布式并行计算和本地计算。这三者共同合作，完成并行分布式计算的任务。

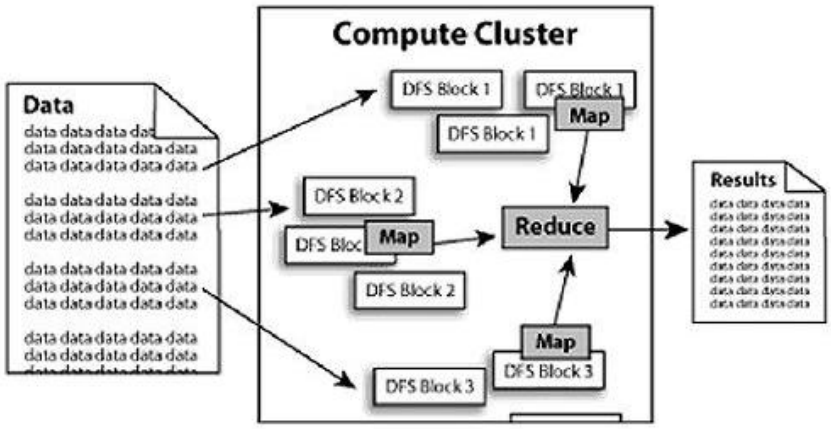

图 4-6 分布存储与并行计算

4.5.1 数据分布存储

如图 4-7 所示，Hadoop 中的分布式文件系统 HDFS 由一个管理节点(NameNode)和 N 个数据节点(DataNode)组成，每个节点均是一台普通的计算机。在使用上同我们熟悉的单机上的文件系统非常类似，一样可以建目录、创建、复制、删除文件、查看文件内容等。但其底层实现上是把文件切割成 Block，然后这些 Block 分散地存储于不同的 DataNode 上，每个 Block 还可以复制数份存储于不同的 DataNode 上，达到容错容灾之目的。NameNode 则是整个 HDFS 的核心，它通过维护一些数据结构，记录了每一个文件被切割成了多少个 Block，这些 Block 可以从哪些 DataNode 中获得，各个 DataNode 的状态等重要信息。

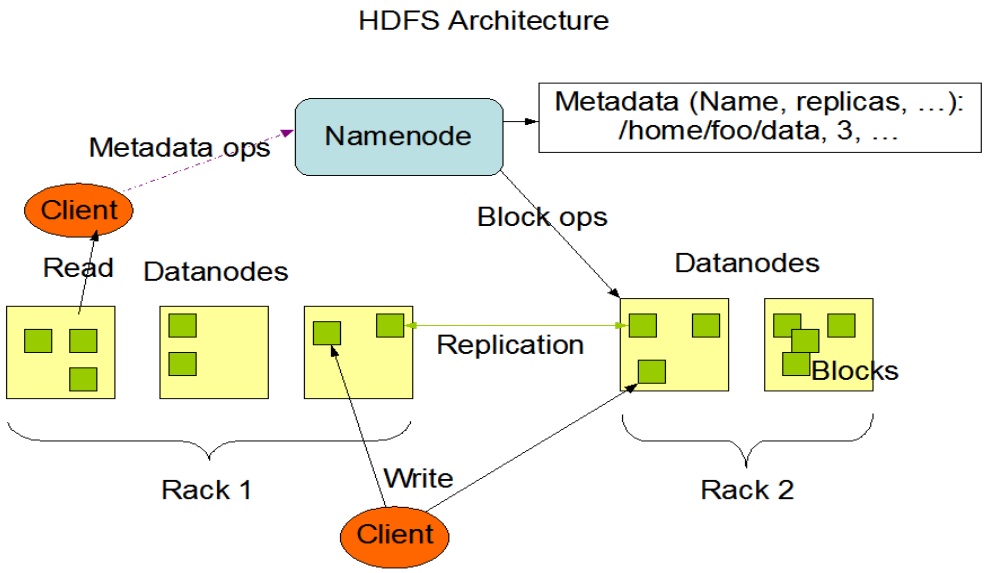

图 4-7 HDFS 的体系结构

4.5.2 分布式并行计算

Hadoop 中有一个作为主控的 JobTracker（见图 4-8），用于调度和管理其它的 TaskTracker，JobTracker 可以运行于集群中任一台计算机上。TaskTracker 负责执行任务，必须运行于 DataNode 上，即 DataNode 既是数据存储节点，也是计算节点。JobTracker 将 Map 任务和 Reduce 任务分发给空闲的 TaskTracker，让这些任务并行运行，并负责监控任务的运行情况。如果某一个 TaskTracker 出故障了，JobTracker 会将其负责的任务转交给另一个空闲的 TaskTracker 重新运行。

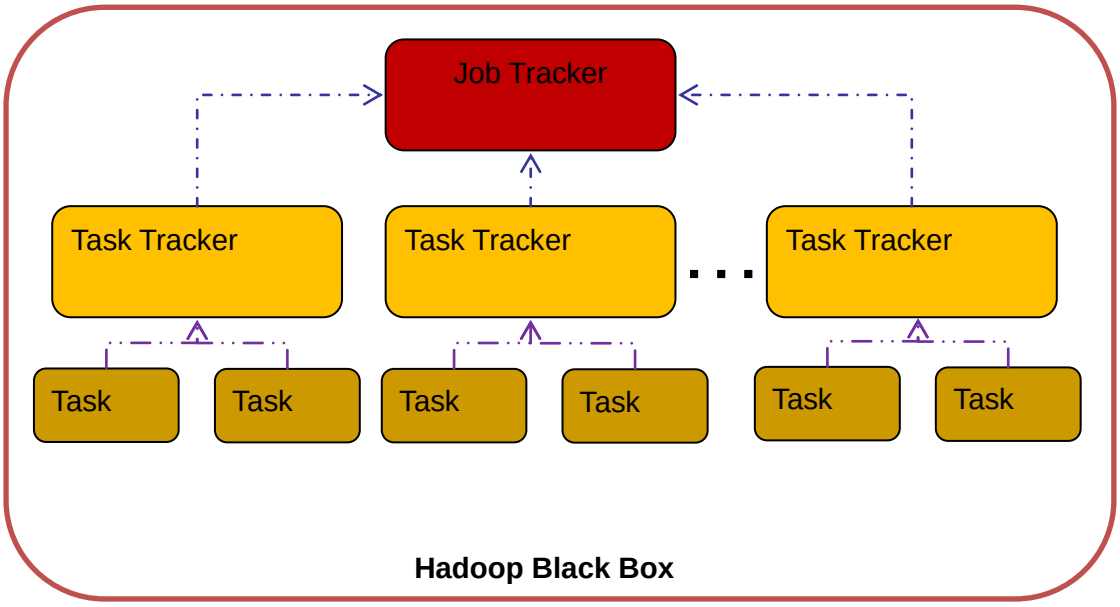

图 4-8 Hadoop Job Tracker

在进行 MapReduce 的任务调度时，首先要保证 master 节点的 NameNode、SecondaryNameNode、JobTracker 和 slaves 节点的 DataNode、TaskTracker 都已经启动。通常来说，MapRedcue 作业是通过 JobClient.rubJob(job)方法向 master 节点的 JobTracker 提交的，JobTracker 接到 JobClient 的请求后把其加入作业队列中。JobTracker 一直在等待 JobClient 通过 RPC 向其提交作业，而 TaskTracker 一直通过 RPC 向 JobTracker 发送心跳信号询问有没有任务可做，如果有，则请求 JobTracker 派发任务给它执行。如果 JobTracker 的作业队列不为空，则 TaskTracker 发送的心跳将会获得 JobTracker 给它派发的任务。这是一个主动请求的任务，slave 的 TaskTracker 主动向 master 的 JobTracker 请求任务。当 TaskTracker 接到任务后，通过自身调度在本 slave 建立起 Task 执行任务。图 4-9 是 MapReduce 任务请求调度的过程示意图。

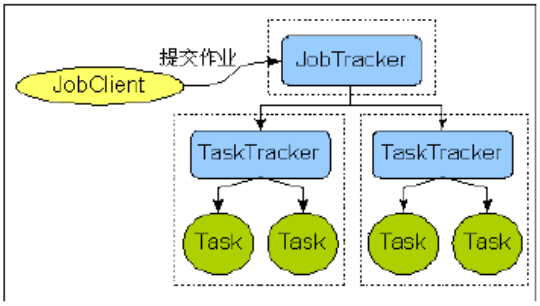

图 4-9 MapReduce 的任务调度

具体来说，MapReduce 任务请求调度过程包括两个步骤：

（1）JobClient 提交作业

JobClient.runJob(job)静态方法会实例化一个 JobClient 的实例，然后用此实例的 submitJob(job)方法向 JobTracker 提交作业。此方法会返回一个 RunningJob 对象，它用来跟踪作业的状态。作业提交完毕后，JobClient 会根据此对象开始关注作业的进度，直到作业完成。submitJob(job)内部是通过调用 submitJobInternal(job)方法完成实质性的作业提交的。submitJobInternal(job)方法首先会向 Hadoop 分布系统文件系统（HDFS）依次上传三个文件 job.jar、job.split 和 job.xml。job.jar 里面包含了执行此任务需要的各种类，比如 Mapper、Reducer 等实现；job.split 是文件分块的相关信息，比如有数据分多少个块、块的大小(默认 64M)等。job.xml 是有关的作业配置，例如 Mapper,Combiner,Reducer 的类型，输入输出格式的类型等。

（2）JobTacker 调度作业

JobTracker 接到 JobClient 提交的作业后，即在 JobTracker.submitJob(job)方法中，首先产

生一个 JobInProgress 对象。此对象代表一道作业，它的作用是维护这道作业的所有信息，包括作业相关信息 JobProfile 和最近作业状态 JobStatus，并将作业所有规划的 Task 登记到任务列表中。随后 JobTracker 将此 JobInProgress 对象通过 listener.jobAdded(job)方法加入到调度队列中，并用一个成员变量 jobs 来维护所有的作业。然后等到有 TaskTracker 空闲，使用 JobTracker.AssignTask(tasktracker）来请求任务，如果调度队列不空，程序便通过调度算法取出一个 task 交给来请求的 TaskTracker 去执行。至此，整个任务分配过程基本完成。

4.5.3 本地计算

数据存储在哪一台计算机上，就由这台计算机进行这部分数据的计算，这样可以减少数据在网络上的传输，降低对网络带宽的需求。在 Hadoop 这样的基于集群的分布式并行系统中，计算节点可以很方便地扩充，因它所能够提供的计算能力近乎是无限的。但是，由是数据需要在不同的计算机之间流动，故而网络带宽变成了瓶颈，是非常宝贵的，因此，“本地计算”是最有效的一种节约网络带宽的手段，业界把这形容为“移动计算比移动数据更经济”。

4.5.4 任务粒度

把原始大数据集切割成小数据集时，通常让小数据集小于或等于 HDFS 中一个 Block 的大小(缺省是 64M)，这样能够保证一个小数据集位于一台计算机上，便于本地计算。有 M 个小数据集待处理，就启动 M 个 Map 任务，注意这 M 个 Map 任务分布于 N 台计算机上并行运行，Reduce 任务的数量 R 则可由用户指定。

4.5.5 Partition

Partition 是选择配置，主要作用是在多个 Reduce 的情况下，指定 Map 的结果由某一个 Reduce 处理，每一个 Reduce 都会有单独的输出文件。把 Map 任务输出的中间结果按 key 的范围划分成 R 份(R 是预先定义的 Reduce 任务的个数)，划分时通常使用 hash 函数，如: hash(key) mod R，这样可以保证某一段范围内的 key，一定是由一个 Reduce 任务来处理，可以简化 Reduce 的过程。

4.5.6 Combine

在 partition 之前，还可以对中间结果先做 combine，即将中间结果中有相同 key 的 <key，value>对合并成一对。Combiner 是可选择的，它的主要作用是在每一个 Map 执行完分析以后，在本地优先做 Reduce 的工作，减少在 Reduce 过程中的数据传输量。combine 的过程与 Reduce 的过程类似，很多情况下就可以直接使用 Reduce 函数，但 combine 是作为 Map 任务的一部分，在执行完 Map 函数后紧接着执行的。Combine 能够减少中间结果中<key，value>对的数目，从而减少网络流量。

4.5.7 Reduce 任务从 Map 任务节点取中间结果

Map 任务的中间结果在做完 Combine 和 Partition 之后，以文件形式存于本地磁盘。中间结果文件的位置会通知主控 JobTracker，JobTracker 再通知 Reduce 任务到哪一个 DataNode 上去取中间结果。注意所有的 Map 任务产生中间结果均按其 Key 用同一个 Hash 函数划分成了 R 份，R 个 Reduce 任务各自负责一段 Key 区间。每个 Reduce 需要向许多个 Map 任务节点取得落在其负责的 Key 区间内的中间结果，然后执行 Reduce 函数，形成一个最终的结果文件。

4.5.8 任务管道

有 R 个 Reduce 任务，就会有 R 个最终结果，很多情况下这 R 个最终结果并不需要合并成一个最终结果。因为这 R 个最终结果又可以做为另一个计算任务的输入，开始另一个并行计算任务。

4.6 实例分析：WordCount

如果想统计过去 10 年计算机论文中出现次数最多的几个单词，看看大家都在研究些什么，那收集好论文后，该怎么办呢？可以大致采用以下几种方法：

- 方法一：可以写一个小程序，把所有论文按顺序遍历一遍，统计每一个遇到的单词的出现次数，最后就可以知道哪几个单词最热门了。这种方法在数据集比较小时，是非常有效的，而且实现最简单，用来解决这个问题很合适。

- 方法二：写一个多线程程序，并发遍历论文。这个问题理论上是可以高度并发的，因为统计一个文件时不会影响统计另一个文件。当我们的机器是多核或者多处理器，方法二肯定比方法一高效。但是，写一个多线程程序要比方法一困难多了，我们必须自己同步共享数据，比如要防止两个线程重复统计文件。
- 方法三：把作业交给多个计算机去完成。我们可以使用方法一的程序，部署到 N 台机器上去，然后把论文集分成 N 份，一台机器跑一个作业。这个方法跑得足够快，但是部署起来很麻烦，我们要人工把程序复制分发到别的机器，要人工把论文集分开，最痛苦的是还要把 N 个运行结果进行整合（当然我们也可以再写一个程序）。
- 方法四：让 MapReduce 来帮帮我们吧！MapReduce 本质上就是方法三，但是如何拆分文件集，如何复制分发程序，如何整合结果，这些都是框架定义好的。我们只要定义好这个任务（用户程序），其它都交给 MapReduce 去处理。

4.6.1 WordCount 设计思路

WordCount 例子如同 Java 中的“HelloWorld”经典程序一样是 MapReduce 的入门程序。计算出文件中各个单词的频数，要求输出结果按照单词的字母顺序进行排序，每个单词和其频数占一行，单词和频数之间有间隔。比如，输入一个文件，其内容如下：

hello world

hello hadoop

hello mapreduce

对应上面给出的输入样例，其输出样例为：

hadoop 1

hello 3

mapreduce 1

world 1

上面这个应用实例的解决方案很直接，就是将文件内容切分成单词。然后将所有相同的单词聚集到一起，最后，计算单词出现的次数进行输出。针对 MapReduce 并行程序设计原则可知，解决方案中的内容切分步骤和数据不相关，可以并行化处理，每个拿到原始数据的机器只要将输入数据切分成单词就可以了。所以，可以在 map 阶段完成单词切分的任务。另外，相同单词的频数计算也可以并行化处理。根据实例要求来看，不同单词之间的频数不

相关，所以，可以将相同的单词交给一台机器来计算频数，然后输出最终结果。这个过程可以交给 reduce 阶段完成。至于将中间结果根据不同单词进行分组后再发送给 reduce 机器，这正好是 MapReduce 过程中的 shuffle 能够完成的。至此，这个实例的 MapReduce 程序就设计出来的。Map 阶段完成由输入数据到单词切分的工作，shuffle 阶段完成相同单词的聚集和分发工作（这个过程是 MapReduce 的默认过程，不用具体配置），reduce 阶段完成接收所有单词并计算其频数的工作。由于 MapReduce 中传递的数据都是<key, value>形式的，并且 shuffle 排序聚集分发是按照 key 值进行的，所以，将 map 的输出设计成由 word 作为 key，1 作为 value 的形式，它表示单词出现了 1 次（map 的输入采用 Hadoop 默认的输入方式，即文件的一行作为 value，行号作为 key）。Reduce 的输入是 map 输出聚集后的结果，即<key, value-list>，具体到这个实例就是<word, {1,1,1,1,…}>，reduce 的输出会设计成与 map 输出相同的形式，只是后面的数值不再是固定的 1，而是具体算出的 word 所对应的频数。

4.6.2 WordCount 代码

采用 MapReduce 进行词频统计的 WordCount 代码如下：

```
public class WordCount
{
    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>
    {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(Object key, Text value, Context context ) throws IOException,
InterruptedException
        {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens())
            {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer extends Reducer<Text, IntWritable, Text,
IntWritable>
    {
```

```
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,Context context)
throws IOException, InterruptedException
        {
            int sum = 0;
            for (IntWritable val : values)
            {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception
    {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 2)
        {
            System.err.println("Usage: wordcount <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

4.6.3 过程解释

map 操作的输入是<key, value>形式，其中，key 是文档中某行的行号，value 是该行的内容。map 操作会将输入文档中每一个单词的出现输出到中间文件中去（如图 4-10 所示）。

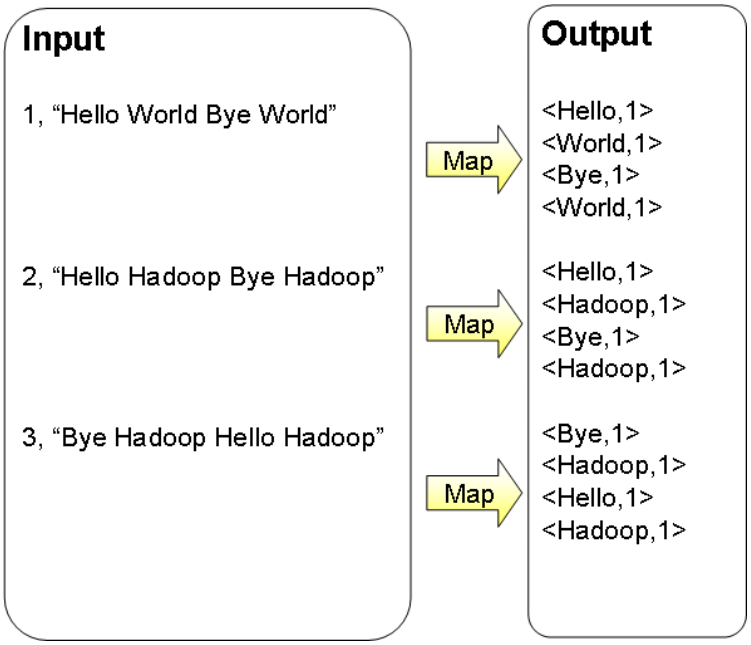

图 4-10 Map 过程示意图

Reduce 操作的输入是单词和出现次数的序列（如图 4-11 所示）。用上面的例子来说，就是 <“Hello”， [1,1,1]>， <“World”，[1,1]>， <“Bye”，[1,1,1]>， <“Hadoop”， [1,1,1,1]> 等。然后根据每个单词，算出总的出现次数。

最后输出排序后的最终结果就会是：<“Bye”，3>，<“Hadoop”，4> ，<“Hello”，3>，<“World”，2>。

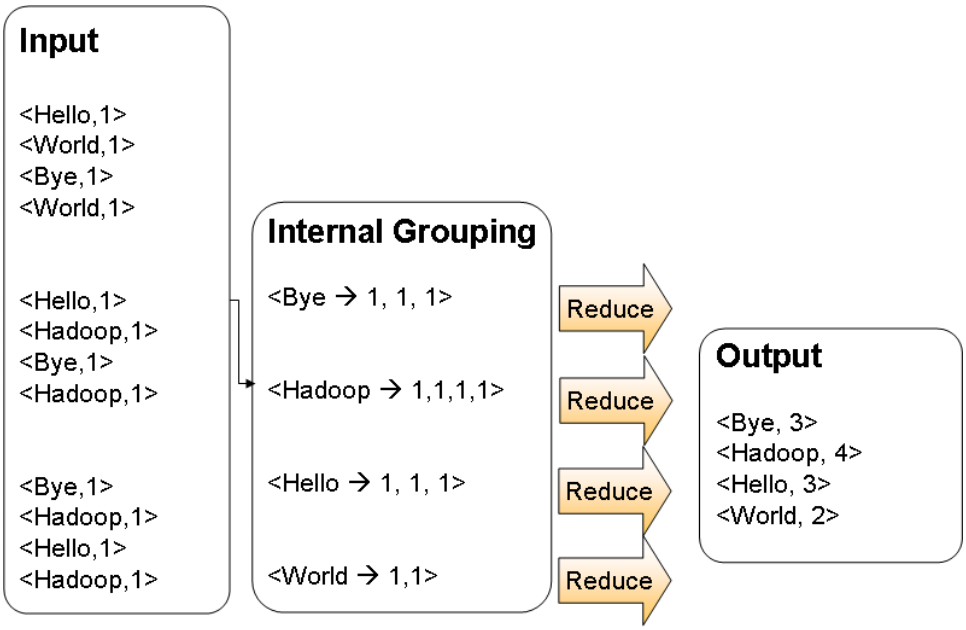

图 4-11 Reduce 过程示意图

整个 MapReduce 过程实际的执行顺序是：

- MapReduce Library 将 Input 分成 M 份，这里的 Input Splitter 也可以是多台机器并

行 Split；

- Master 将 M 份 Job 分给空闲状态的 M 个 worker 来处理；
- 对于输入中的每一个<key， value> 进行 Map 操作，将中间结果缓冲在内存里；
- 定期地（或者根据内存状态）将缓冲区中的中间信息刷写到本地磁盘上，并且把文件信息传回给 Master（Master 需要把这些信息发送给 Reduce worker）。这里最重要的一点是，在写磁盘的时候，需要将中间文件做 Partition（比如 R 个）。拿上面的例子来举例，如果把所有的信息存到一个文件，Reduce worker 又会变成瓶颈。我们只需要保证相同 Key 能出现在同一个 Partition 里面就可以把这个问题分解。
- R 个 Reduce worker 开始工作，从不同的 Map worker 的 Partition 那里拿到数据，用 key 进行排序（如果内存中放不下需要用到外部排序）。很显然，排序（或者说 Group）是 Reduce 函数之前必须做的一步。这里面很关键的是，每个 Reduce worker 会去从很多 Map worker 那里拿到 X(0<X<R) Partition 的中间结果，这样，所有属于这个 Key 的信息已经都在这个 worker 上了。
- Reduce worker 遍历中间数据，对每一个唯一 Key，执行 Reduce 函数（参数是这个 key 以及相对应的一系列 Value）。
- 执行完毕后，唤醒用户程序，返回结果（最后应该有 R 份 Output，每个 Reduce Worker 一个）。

可见，这里的“分”（Divide）体现在两步，分别是将输入分成 M 份，以及将 Map 的中间结果分成 R 份。将输入分开通常很简单，而把 Map 的中间结果分成 R 份，通常用”hash(key) mod R”这个结果作为标准，保证相同的 Key 出现在同一个 Partition 里面。当然，使用者也可以指定自己的 Partition 函数，比如，对于 Url Key，如果希望同一个 Host 的 URL 出现在同一个 Partition，可以用”hash(Hostname(urlkey)) mod R”作为 Partition 函数。

另外，对于上面的例子来说，每个文档中都可能会出现成千上万的 <“the”，1>这样的中间结果，琐碎的中间文件必然导致传输上的开销。因此，MapReduce 还支持用户提供 Combiner 函数。这个函数通常与 Reduce 函数有相同的实现，不同点在于 Reduce 函数的输出是最终结果，而 Combiner 函数的输出是 Reduce 函数的输入。图 4-10 中的 map 过程输出结果，如果采用 Combiner 函数后，则可以得到如图 4-12 所示的输出，这个输出结果可以作为 reduce 过程的输入（如图 4-13 所示）。

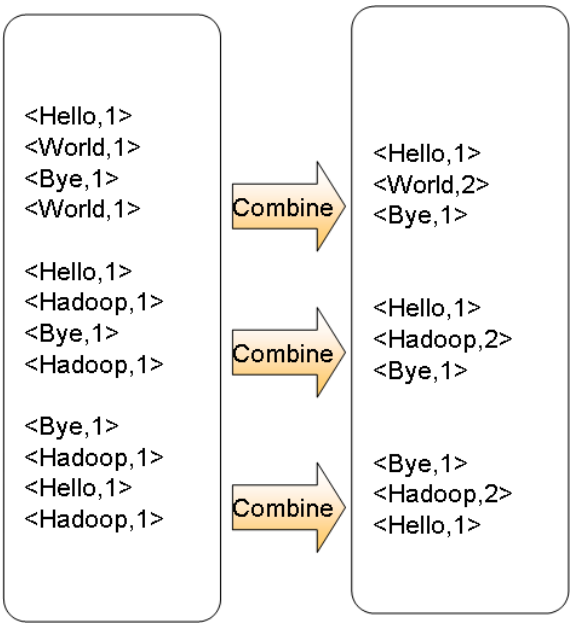

图 4-12 Combine 过程示意图

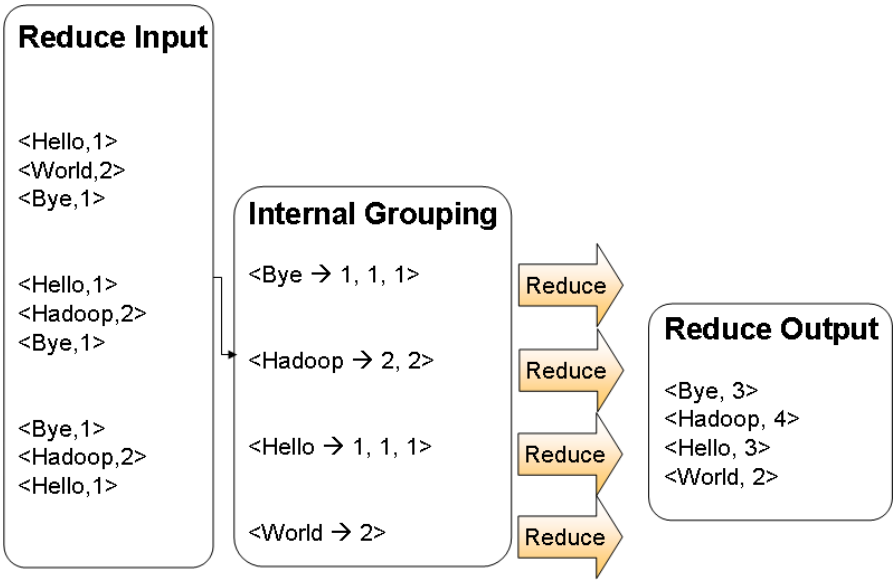

图 4-13 以 Combine 输出结果作为输入的 Reduce 过程示意图

4.7 新 MapReduce 框架 Yarn

Yarn 带来了巨大的改变，改变了 Hadoop 计算组件（MapReduce）切分和重新组成处理任务的方式，因为，Yarn 把 MapReduce 的追踪组件切分成两个不同部分：资源管理器和应用调度。这样的数据管理工具，有助于更加轻松地同时运行 MapReduce 或 Storm 这样的任务以及 HBase 等服务。Hadoop 共同创始人之一 Doug Cutting 表示：“它使得其他不是 MapReduce 的工作负载现在可以更有效地与 MapReduce 分享资源。现在这些系统可以动态地分享资源，资源也可以设置优先级”。

Cutting 和 Bhandarkar 都承认，这种方法是受到了 Apache 项目“Mesos”集群管理系统以及谷歌 Borg 和 Omega 秘密项目的一些影响。

Yarn 的出现，使得 Hadoop 变成一个针对数据中心的操作系统，支持广泛的应用。

4.7.1 原 Hadoop MapReduce 框架的问题

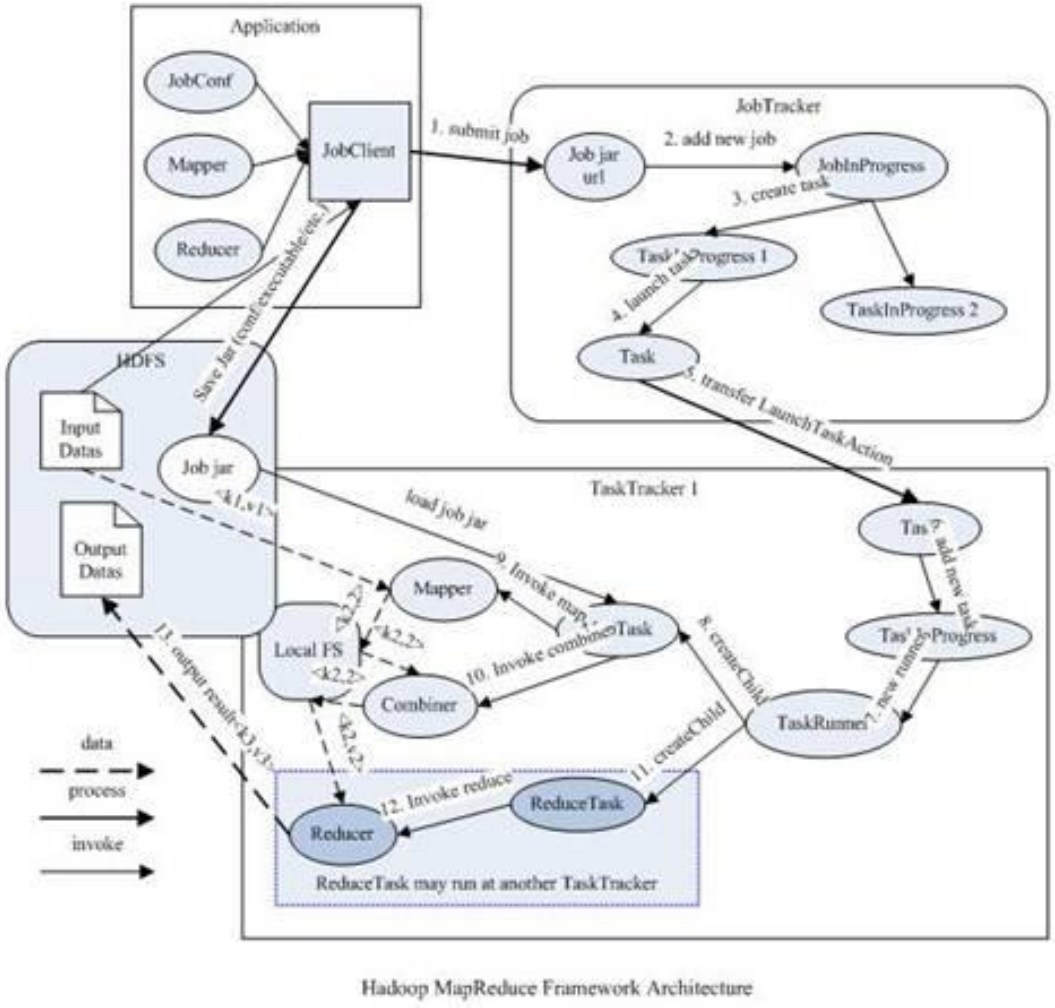

图 4-14 原 Hadoop MapReduce 架构

从图 4-14 中可以清楚地看出原 MapReduce 程序的流程及设计思路：

- 首先用户程序 (JobClient) 提交了一个 job，job 的信息会发送到 Job Tracker 中，Job Tracker 是 MapReduce 框架的中心，它需要与集群中的机器定时通信 (heartbeat)，需要管理哪些程序应该跑在哪些机器上，需要管理所有 job 失败、重启等操作。
- TaskTracker 是 MapReduce 集群中每台机器都有的一个部分，它做的事情主要是监视自己所在机器的资源情况。
- TaskTracker 同时监视当前机器的 tasks 运行状况。TaskTracker 需要把这些信息通

过 heartbeat 发送给 JobTracker，JobTracker 会搜集这些信息以确定新提交的 job 分配运行在哪些机器上。图 4-14 虚线箭头就是表示消息的“发送—接收”的过程。

可以看得出，原来的 MapReduce 架构是简单明了的，在最初推出的几年，也得到了众多的成功案例，获得业界广泛的支持和肯定。但是，随着分布式系统集群的规模和其工作负荷的增长，原框架的问题逐渐浮出水面，主要的问题集中如下：

- JobTracker 是 MapReduce 的集中处理点，存在单点故障。
- JobTracker 完成了太多的任务，造成了过多的资源消耗，当 MapReduce job 非常多的时候，会造成很大的内存开销，潜在来说，也增加了 JobTracker 失败的风险，这也是业界普遍总结出来的一个结论，即老 Hadoop 的 MapReduce 只能支持 4000 节点主机的上限。
- 在 TaskTracker 端，以 MapReduce 任务的数目作为资源的表示过于简单，没有考虑到 CPU 和内存的占用情况，如果两个大内存消耗的任务被调度到了一块，很容易出现 OOM（Out of Memory）。
- 在 TaskTracker 端，把资源强制划分为 map task slot 和 reduce task slot, 如果当系统中只有 map task 或者只有 reduce task 的时候，会造成资源的浪费，也就是前面提过的集群资源利用的问题。
- 源代码层面分析的时候，会发现代码非常难读，常常因为一个类（class）做了太多的事情，代码量达 3000 多行，造成类的任务不清晰，增加 bug 修复和版本维护的难度。
- 从操作的角度来看，现在的 Hadoop MapReduce 框架在有任何重要的或者不重要的变化时(例如 bug 修复，性能提升和特性化)，都会强制进行系统级别的升级更新。更糟的是，它不管用户的喜好，强制让分布式集群系统的每一个用户端同时更新。这些更新会让用户为了验证他们之前的应用程序是不是适用新的 Hadoop 版本而浪费大量时间。

4.7.2 新 Hadoop Yarn 框架原理及运作机制

从业界使用分布式系统的变化趋势和 Hadoop 框架的长远发展来看，MapReduce 的 JobTracker/TaskTracker 机制需要大规模的调整来修复它在可扩展性、内存消耗、线程模型、可靠性和性能上的缺陷。在过去的几年中，Hadoop 开发团队做了一些 bug 的修复，但是，

最近这些修复的成本越来越高，这表明对原框架做出改变的难度越来越大。

为从根本上解决旧 MapReduce 框架的性能瓶颈，促进 Hadoop 框架的更长远发展，从 0.23.0 版本开始，Hadoop 的 MapReduce 框架完全重构，发生了根本的变化。新的 Hadoop MapReduce 框架命名为 MapReduceV2 或者叫 Yarn，其架构图如图 4-15 所示。

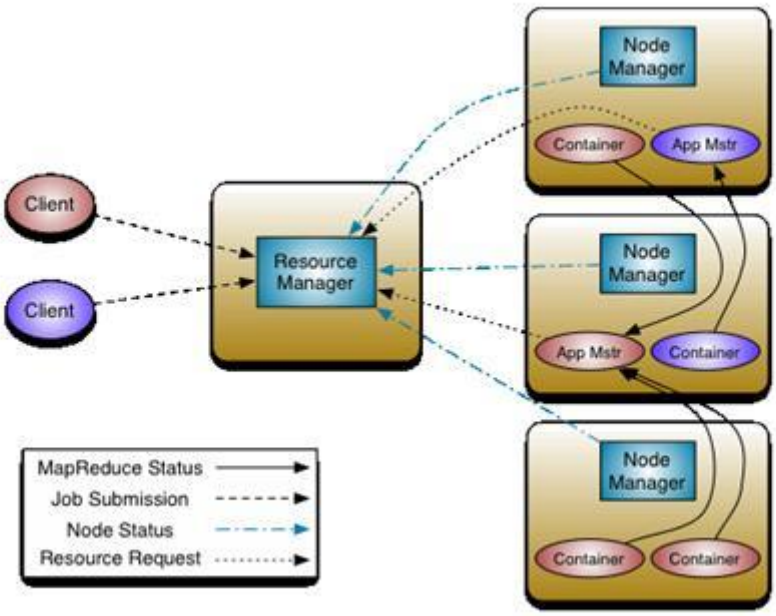

图 4-15 新的 Hadoop MapReduce 框架（Yarn）架构

重构根本的思想是将 JobTracker 两个主要的功能分离成单独的组件，这两个功能是资源管理和任务调度/监控。新的资源管理器全局管理所有应用程序计算资源的分配，每一个应用的 ApplicationMaster 负责相应的调度和协调。一个应用程序无非是一个单独的传统的 MapReduce 任务或者是一个 DAG(有向无环图) 任务。ResourceManager 和每一台机器的节点管理服务器能够管理用户在那台机器上的进程并能对计算进行组织。

事实上，每一个应用的 ApplicationMaster 是一个详细的框架库，它结合从 ResourceManager 获得的资源，和 NodeManager 协同工作来运行和监控任务。

图 4-15 中 ResourceManager 支持分层级的应用队列，这些队列享有集群一定比例的资源。从某种意义上讲，它就是一个纯粹的调度器，它在执行过程中不对应用进行监控和状态跟踪。同样，它也不能重启因应用失败或者硬件错误而运行失败的任务。

ResourceManager 是基于应用程序对资源的需求进行调度的；每一个应用程序需要不同类型的资源，因此就需要不同的容器。资源包括内存、CPU、磁盘、网络等等。可以看出，这同原来的 MapReduce 中固定类型的资源使用模型有显著区别，原来的那种模型给集群的使用会带来负面的影响。资源管理器提供一个调度策略的插件，它负责将集群资源分配给多

个队列和应用程序。调度插件可以基于现有的能力调度和公平调度模型。

图 4-15 中 NodeManager 是每一台机器框架的代理，是执行应用程序的容器，监控应用程序的资源使用情况 (CPU、内存、硬盘、网络) 并且向调度器汇报。

每一个应用的 ApplicationMaster 的职责有：向调度器索要适当的资源容器，运行任务，跟踪应用程序的状态和监控它们的进程，处理任务的失败原因。

4.7.3 新旧 Hadoop MapReduce 框架比对

（1）新旧 MapReduce 框架的变化

新的 Yarn 框架相对旧 MapRduce 框架而言，其配置文件、启停脚本及全局变量等发生了一些变化。

新旧 MapReduce 框架的客户端没有发生变化，其调用 API 及接口，大部分保持兼容，这也是为了对开发使用者透明化，使其不必对原有代码做大的改变。但是，原框架中核心的 JobTracker 和 TaskTracker 不见了，取而代之的是 ResourceManager、ApplicationMaster 与 NodeManager 三个部分，三者的具体功能如下：

- **ResourceManager**：ResourceManager 是一个中心的服务，它做的事情是调度、启动每一个 Job 所属的 ApplicationMaster，并且监控 ApplicationMaster 的存在情况。细心的读者会发现：Job 里面所在的 task 的监控、重启等等内容不见了。这就是 ApplicationMaster 存在的原因。ResourceManager 负责作业与资源的调度。接收 JobSubmitter 提交的作业，按照作业的上下文(Context) 信息以及从 NodeManager 收集来的状态信息，启动调度过程，分配一个 Container 作为 ApplicationMaster。
- **NodeManager**：功能比较专一，就是负责 Container 状态的维护，并向 ResourceManager 保持心跳。
- **ApplicationMaster**：负责一个 Job 生命周期内的所有工作，类似老的框架中 JobTracker。但是要注意，每一个 Job（不是每一种）都有一个 ApplicationMaster，它可以运行在 ResourceManager 以外的机器上。

（2）Yarn 框架相对于老的 MapReduce 框架的优势

Yarn 框架相对于老的 MapReduce 框架什么优势呢？具体表现在以下几个方面：

- 这个设计大大减小了 JobTracker（也就是现在的 ResourceManager）的资源消耗，并且让监测每一个 Job 子任务 (tasks) 状态的程序分布式化了，更安全、更优雅。
- 在新的 Yarn 中，ApplicationMaster 是一个可变更的部分，用户可以对不同的编程模型写自己的 ApplicationMaster，让更多类型的编程模型能够跑在 Hadoop 集群中，可以参考 Hadoop Yarn 官方配置模板中的 mapred-site.xml 配置。
- 对于资源的表示以内存为单位 (在目前版本的 Yarn 中，没有考虑 CPU 的占用)，比之前以剩余 slot 数目更合理。
- 老的框架中，JobTracker 一个很大的负担就是监控 job 下的 tasks 的运行状况，现在，这个部分就扔给 ApplicationMaster 做了，而 ResourceManager 中有一个模块叫做 ApplicationsMasters(注意不是 ApplicationMaster)，它是监测 ApplicationMaster 的运行状况，如果出问题，会将其在其他机器上重启。
- Container 是 Yarn 为了将来作资源隔离而提出的一个框架。这一点应该借鉴了 Mesos 的工作，目前是一个框架，仅仅提供 java 虚拟机内存的隔离，Hadoop 团队的设计思路应该后续能支持更多的资源调度和控制，既然资源表示成内存量，那就没有了之前的 map slot/reduce slot 分开造成集群资源闲置的尴尬情况。

本章小结

本章介绍了 MapReduce 计算模型的基本原理、工作流程，阐述了 Hadoop 中实现并行计算的相关机制以及 MapReduce 的任务调度过程；接下来，以一个实例演示了 MapReduce 的详细执行过程；最后，介绍了新 MapReduce 框架 Yarn 的原理及运作机制。

参考文献

[1] MapReduce. 百度百科.

[2] 分布式计算开源框架 Hadoop 入门实践. http://www.chinacloud.cn/show.aspx?id=463&cid=12

[3] Dean, Jeffrey & Ghemawat, Sanjay (2004). "MapReduce: Simplified Data Processing on Large Clusters". Retrieved Apr. 6, 2005

[4] 陆嘉恒. Hadoop 实战. 机械工业出版社. 2011 年.

[5] 其他网络来源.

第 5 章 HDFS

Hadoop 分布式文件系统 HDFS（Hadoop Distributed File System）是一个能够兼容普通硬件环境的分布式文件系统，和现有的分布式文件系统不同的地方是，Hadoop 更注重容错性和兼容廉价的硬件设备，这样做是为了用很小的预算甚至直接利用现有机器就实现大流量和大数据量的读取。Hadoop 使用了 POSIX 的设计来实现对文件系统文件流的读取。HDFS 原来是 Apache Nutch 搜索引擎（从 Lucene 发展而来）开发的一个部分，后来独立出来作为一个 Apache 子项目。

本章介绍 HDFS 的相关知识，内容要点如下：

- HDFS 的假设与目标
- HDFS 的相关概念
- HDFS 体系结构
- HDFS 命名空间
- HDFS 存储原理
- 通讯协议
- 数据错误与异常
- 从 HDFS 看分布式文件系统的设计需求

5.1HDFS 的假设与目标

HDFS 是基于流数据模式访问和处理超大文件的需求而开发的，它可以运行于廉价的商用服务器上。HDFS 在设计时的假设和目标包括以下几个方面：

- **硬件出错**：Hadoop 假设硬件出错是一种正常的情况，而不是异常，为的就是在硬件出错的情况下尽量保证数据完整性，HDFS 设计的目标是在成百上千台服务器中存储数据，并且可以快速检测出硬件错误和快速进行数据的自动恢复。
- **流数据读写**：不同于普通的文件系统，Hadoop 是为了程序批量处理数据而设计的，而不是与用户的交互或者随机读写，所以 POSIX 对程序增加了许多硬性限制，程序必须使用流读取来提高数据吞吐率。
- **大数据集**：HDFS 上面一个典型的文件一般是用 GB 或者 TB 计算的，而且一个数

百台机器组成的集群里面可以支持过千万这样的文件。

- **简单的文件模型**：HDFS 上面的文件模型十分简单，就是一次写入多次读取的模型，文件一旦创建，写入并关闭了，之后就再也不会被改变了，只能被读取，这种模型刚好符合搜索引擎的需求，以后可能会实现追加写入数据这样的功能。
- **强大的跨平台兼容性**：由于是基于 Java 的实现，无论是硬件平台或者是软件平台要求都不高，只要是 JDK 支持的平台都可以兼容。

正是由于以上的种种考虑，我们会发现，现在的 HDFS 在处理一些特定问题时，不但没有优势，而且有一定的局限性，主要表现在以下几个方面：

- **不适合低延迟数据访问**：如果要处理一些用户要求时间比较短的低延迟应用请求，则 HDFS 不适合。HDFS 是为了处理大型数据集分析任务，主要是为了达到较高的数据吞吐量而设计的，这就可能以高延迟作为代价。目前的一些补充的方案，比如使用 HBase，通过上层数据管理项目来尽可能地弥补这个不足。
- **无法高效存储大量小文件**：在 Hadoop 中需要使用 NameNode（目录节点）来管理文件系统的元数据，以响应客户端请求返回文件位置等，因此，文件数量大小的限制要由 NameNode 来决定。例如，每个文件、索引目录及块大约占 100 字节，如果有 100 万个文件，每个文件占一个块，那么，至少要消耗 200MB 内存，这似乎还可以接受。但是，如果有更多文件，那么，NameNode 的工作压力更大，检索处理元数据的时间就不可接受了。
- **不支持多用户写入及任意修改文件**：在 HDFS 的一个文件中只有一个写入者，而且写操作只能在文件末尾完成，即只能执行追加操作。目前 HDFS 还不支持多个用户对同一文件的写操作，以及在文件任意位置进行修改。

5.2HDFS 的相关概念

5.2.1 块（Block）

我们知道，在操作系统中都有一个文件块的概念，文件以块的形式存储在磁盘中，此处块的大小代表系统读/写可操作的最小文件大小。也就是说，文件系统每次只能操作磁盘块大小的整数倍数据。通常来说，一个文件系统块为几千字节，而磁盘块大小为 512 字节。文件的操作都由系统完成，这些对用户来说都是透明的。

这里，我们所要介绍的 HDFS 的块是抽象的概念，它比上面操作系统中所说的块要大得多。在配置 Hadoop 系统时会看到，它的默认块为 64MB。和单机上的文件系统相同，HDFS 分布式文件系统中的文件也被分成块进行存储，它是文件存储处理的逻辑单元。

HDFS 作为一个分布式文件系统，是设计用来处理大文件的，使用抽象的块可以带来很多好处。一个好处就是，可以存储任意大的文件，而又不会受到网络中任一单个节点磁盘大小的限制。可以想象一下，单个节点存储 100TB 的数据是不可能的，但是，由于逻辑块的设计，HDFS 可以将这个超大的文件分成众多块，分别存储在集群的各台机器上。另外一个好处是使用抽象块作为操作的单元，可以简化存储子系统。这里之所以提到简化，是因为这是所有系统的追求，而对故障出现频繁和种类繁多的分布式系统来说，简化就显得尤为重要。在 HDFS 中块的大小是固定的，这样就简化了存储系统的管理，特别是元数据信息可以和文件块内容分开存储。不仅如此，块更有利于分布式文件系统中复制容错的实现。在 HDFS 中为了处理节点故障，默认将文件块副本数设定为 3 份，分别存储在集群的不同节点上。当一个块损坏时，系统会通过 NameNode 获取元数据信息，在另外的机器上读取一个副本并进行存储，这个过程对用户来说都是透明的。当然，这里的文件块副本冗余量，可以通过文件进行配置，比如在有些应用中，可能会为操作频率较高的文件块设置较高的副本数量以及提高集群的吞吐量。

5.2.2 NameNode 和 DataNode

HDFS 体系结构中有两类节点，一类是 NameNode，另一类是 DataNode。这两类节点分别承担 Master 和 Worker 的任务。

- **目录节点（NameNode）**是集群里面的主节点，负责文件名的维护管理，也是客户端访问文件的入口。文件名的维护包括文件和目录的创建、删除、重命名等。同时也管理数据块和数据节点的映射关系，客户端需要访问目录节点才能知道一个文件的所有数据块都保存在哪些数据节点上。
- **数据节点（DataNode）**一般就是集群里面的一台机器，负责数据的存储和读取。在写入时，由目录节点分配数据块的保存位置，然后客户端直接写到对应的数据节点。在读取时，当客户端从目录节点获得数据块的映射关系后，就会直接到对应的数据节点读取数据。数据节点也要根据目录节点的命令创建、删除数据块和冗余复制。

5.3HDFS 体系结构

Hadoop 文件系统采用主从架构对文件系统进行管理，一个 HDFS 集群由唯一一个目录节点（NameNode）和数个数据节点（DataNodes）组成（如图 5-1 所示）。目录节点是一个中心服务器，负责管理文件系统的名字空间及客户端对文件的访问。集群中的数据节点一般是一个节点运行一个数据节点进程，管理负责它所在节点上的存储。

HDFS 对外表现为一个普通的文件系统，用户可以用文件名去存储和访问文件，而实际上文件是被分成不同的数据块，这些数据块就是存储在数据节点上面。数据节点负责处理文件系统客户端的读/写请求，在目录节点的统一调度下进行数据块的创建、删除和复制。

一个典型的 Hadoop 文件系统集群部署，是由一台性能较好的机器运行目录节点，而集群里面的其它机器每台上面运行一个数据节点。当然一个机器可以运行任意多个数据节点，甚至目录节点和数据节点一起运行，不过这种模式在正式的应用部署中很少使用。

唯一的目录节点的设计大大简化了整个体系结构，目录节点负责 Hadoop 文件系统里面所有元数据的仲裁和存储。这样的设计使数据不会脱离目录节点的控制。

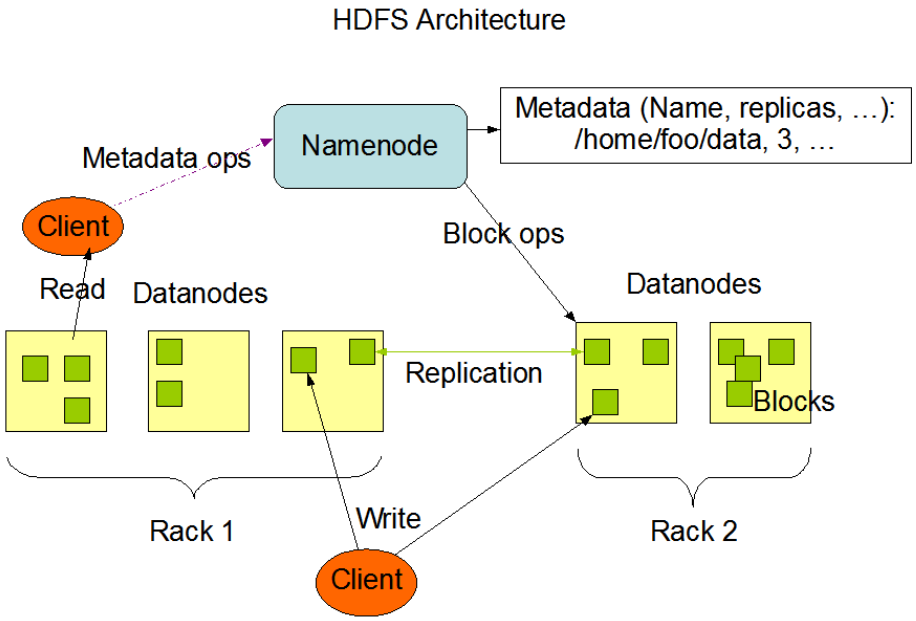

图 5-1 HDFS 的体系结构

5.4HDFS 命名空间

Hadoop 文件系统使用的是传统的分级文件体系，客户端程序可以创建目录并且在目录里面保存文件，类似于现在一般的文件系统。Hadoop 允许用户创建、删除文件，在目录间

转移文件，重命名文件等，但是，还没有实现磁盘配额和文件访问权限等功能，也不支持文件的硬连接和软连接（快捷方式），这些功能在短期内不会实现。

目录节点负责存储和管理整个文件系统的命名空间，任何对文件系统命名空间或属性的修改都将被目录节点记录下来。应用程序可以指定某一个文件需要在 Hadoop 文件系统中冗余多少份，这个在 Hadoop 中称为冗余因子，保存在目录节点里面。

5.5HDFS 存储原理

5.5.1 冗余数据保存

Hadoop 文件系统是为了大文件的可靠保存而设计的，一个文件被划分成一连串的数据块，除了文件的最后一块以外其它所有的数据块都是固定大小的，为了数据容错性，每一个数据块都会被冗余存储起来，即存储多个副本，而每个文件的块大小和冗余参数都是可以设置的，程序可以设置文件的数据块要被复制多少份，而且这个冗余参数除了可以在创建的时候指定，还可以在之后改变。在 Hadoop 文件系统里面文件只会被写入一次，并且任何时间只会有一个程序在写入这个文件。

目录节点是根据数据块的冗余状况来作出处理决策的，数据节点会定期发送一个存在信号（Heartbeat）和数据块列表给目录节点，存在信号使目录节点认为该数据节点还是有效的，而数据块列表包括了该数据节点上面的所有数据块编号。

5.5.2 数据存取策略

复制策略是 Hadoop 文件系统最核心的部分，对读写性能影响很大，Hadoop 和其它分布式文件系统的最大区别就是可以调整冗余数据的位置，这个特性需要很多时间去优化和调整。

- **数据存放**

目前 Hadoop 采用以机柜为基础的数据存放策略，这样做的目的是提高数据可靠性和充分利用网络带宽。当前具体实现了的策略只是这个方向的尝试，Hadoop 短期的研究目标之一就是在实际产品环境中观察系统读写的行为，测试性能和研究更深入的规则。

一个大的 Hadoop 集群经常横跨多个机柜，而不同机柜之间的数据通讯需要经过交换机或者路由，所以，同一个机柜中不同机器之间的通讯带宽，要比不同机柜之间机器的通讯带

宽大。

Hadoop 提供了一个 API 来决定数据机所属的机柜 ID，当文件系统启动的时候，数据机就把自己所属的机柜 ID 发给目录机，然后目录机管理这些分组。

Hadoop 默认是每个数据机都是在不同的机柜上面，这种方法没有做任何性能优化，但是也有不少优点：

- 数据可靠性是最高的，因为这样可以防止机柜出错的时候数据丢失；
- 在读取数据的时候充分利用不同机柜之间的带宽；
- 而且这个策略可以很容易地完成负载平衡和错误处理。

缺点就是写入数据的时候并不能完全利用同一机柜里面机器的带宽。

在默认的配置下，Hadoop 的冗余复制因子是 3，意思就是每一块文件数据一共有 3 个地方存放，Hadoop 目前的存放策略是其中两份放在同一个 rack id 的不同机器上面，另外一个放在不同 rack id 的机器上面，简单来说就是 1/3 的冗余数据在一个机柜里面，2/3 的冗余数据在另外一个机柜里面，这样既可以防止机柜异常时候的数据恢复，又可以提高读写性能。

- **数据读取**

数据读取策略是：根据前面所说的数据存放策略，数据读取的时候，客户端也有 api 确定自己的机柜 id，读取的时候，如果有块数据和客户端的机柜 id 一样，就优先选择该数据节点，客户端直接和数据节点建立连接，读取数据。如果没有，就随机选取一个数据节点。

- **数据复制**

数据复制主要是在数据写入和数据恢复的时候发生，数据复制是使用流水线复制的策略。当客户端要在 Hadoop 上面写一个文件，首先它先把这个文件写在本地，然后对文件进行分块，默认 64MB 一块，每块数据都对 Hadoop 目录服务器发起写入请求，目录服务器选择一个数据机列表，返回给客户端，然后客户端就把数据写入第一台数据机，并且把列表传给数据机，当数据机接收到 4KB 数据的时候，写入本地，并且发起连接到下一台数据机，把这个 4KB 数据传过去，形成一条流水线。当最后文件写完的时候，数据复制也同时完成，这个就是流水线处理的优势。

5.6通讯协议

Hadoop 的通讯协议基本是在 TCP/IP 的基础上开发的，客户端使用 ClientProtocol 和目录服务器通讯，数据机使用 DatanodeProtocol 和目录服务器通讯，而目录服务器一般只是应答

客户端和数据机的请求，不会主动发起通讯。

5.7数据错误和异常

Hadoop 文件系统的主要目标就是在硬件出错的时候保证数据的完整性，它把磁盘错误作为肯定会出现的情况来对待，而不是异常。一般数据存储中出现的错误有几种，分别是目录服务器错误、数据机错误和网络传输异常。

- **数据机出错**

每个数据机会定时发送一个心跳信息给目录服务器，表明自己仍然存活，网络异常可能会导致一部分数据机无法和目录服务器通讯，这时候目录服务器收不到心跳信息，就认为这个数据机已经死机，从有效 I/O 列表中清除，而该数据机上面的所有数据块也会标记为不可读。这个时候某些数据块的冗余份数有可能就低于它的冗余因子了，目录服务器会定期检查每一个数据块，看看它是否需要进行数据冗余复制。

- **出现数据异常**

由于网络传输和磁盘出错的原因，从数据机读取的数据有可能出现异常，客户端会采用 md5 和 sha1 对数据块进行校验。客户端在创建文件的时候，会对每一个文件块进行信息摘录，并把这些信息写入到同一个路径的隐藏文件里面。当客户端读取文件的时候，会先读取该信息文件，然后对每个读取的数据块进行校验，如果校验出错，客户端就会请求到另外一个数据机读取该文件块，并且报告给目录服务器这个文件块有错误，目录服务器就会定期检查，并且重新复制这个块。

- **目录服务器出错**

FsImage 和 Editlog 是目录服务器上面两个最核心的数据结构，如果其中一个文件出错的话，会造成目录服务器不起作用。由于这两个文件非常重要，所以，目录服务器上面可以设置多个备份文件和辅助服务器，当这两个文件有改变的时候，目录服务器就会发起同步操作，虽然这样增加了系统的负担，但是，对于目前这个架构而言，为了实现数据的可靠性，这个同步操作是非常必要的。

5.8从 HDFS 看分布式文件系统的设计需求

分布式文件系统的设计目标大概包括：透明性、并发控制、可伸缩性、容错以及安全需求等。从这几个角度去观察 HDFS 的设计和实现，可以更清楚地看出 HDFS 的应用场景和

设计理念。

- **透明性**

按照开放分布式处理的标准，一般包括 8 种透明性：访问的透明性、位置的透明性、并发透明性、复制透明性、故障透明性、移动透明性、性能透明性和伸缩透明性。对于分布式文件系统，最重要的是希望能达到 5 个透明性要求：

（1）访问的透明性

用户能通过相同的操作来访问本地文件和远程文件资源，HDFS 可以做到这一点。如果把 HDFS 设置成本地文件系统，而非分布式文件系统，那么，读写分布式 HDFS 的程序可以不用修改地读写本地文件，要做修改的只是配置文件。可见，HDFS 提供的访问透明性是不完全的，毕竟它构建于 Java 之上，不能像 NFS 或者 AFS 那样去修改 unix 内核，同时将本地文件和远程文件以一致的方式处理。

（2）位置的透明性

使用单一的文件命名空间，在不改变路径名的前提下，文件或者文件集合可以被重定位。HDFS 集群只有一个目录节点来负责文件系统命名空间的管理，文件的块可以重新分布复制，块可以增加或者减少副本，副本可以跨机架存储，而这一切对客户端都是透明的。

（3）移动的透明性

这一点与位置的透明性类似，HDFS 中的文件经常由于节点的失效、增加或者复制因子的改变或者重新均衡等原因而进行着复制或者移动，而客户端和客户端程序并不需要改变什么，目录节点的日志文件记录着这些变更。

（4）性能的透明性和伸缩的透明性

HDFS 的目标就是构建在大规模廉价机器上的分布式文件系统集群，可伸缩性毋庸置疑，至于性能可以参考它首页上的一些测试基准（benchmark）。

- **并发控制**

客户端对于文件的读写不应该影响其他客户端对同一个文件的读写。要想实现近似原生文件系统的单个文件拷贝语义，分布式文件系统需要做出复杂的交互，例如采用时间戳，或者类似回调承诺（类似服务器到客户端的 RPC 回调，在文件更新的时候；回调有两种状态：有效或者取消；客户端通过检查回调承诺的状态，来判断服务器上的文件是否被更新过）。HDFS 并没有这样做，它的机制非常简单，任何时间都只会有一个程序在写入某个文件，这个文件经创建并写入之后不再改变，它的模型是“一次写入多次读取”。这与它的应用场合是一致的，HDFS 的文件大小通常是 MB 至 TB 级的，这些数据不会经常修改，最经常的是

被顺序读取并处理，随机读很少，因此，HDFS 非常适合 MapReduce 框架或者网页爬虫应用。HDFS 文件的大小也决定了它的客户端不能像某些分布式文件系统那样缓存常用到的几百个文件。

- **文件复制功能**

一个文件可以表示为其内容在不同位置的多个拷贝，这样做带来了两个好处：（1）访问同一个文件时，可以从多个服务器中获取，从而改善服务的伸缩性；（2）提高了容错能力，某个副本损坏了，仍然可以从其他服务器节点获取该文件。HDFS 文件的块，为了容错都将被备份，并且可以配置复制因子，默认是 3。副本的存放策略也是很有讲究的，一个放在本地机架的节点，另一个放在同一机架的另一节点，还有一个放在其他机架上。这样可以最大限度地防止因故障导致的副本丢失。不仅如此，HDFS 读文件的时候也将优先选择从同一机架乃至同一数据中心的节点上读取块。

- **硬件和操作系统的异构性**

由于构建在 Java 平台上，HDFS 的跨平台能力毋庸置疑，得益于 Java 平台已经封装好的文件 IO 系统，HDFS 可以在不同的操作系统和计算机上实现同样的客户端和服务端程序。

- **容错能力**

在分布式文件系统中，尽量保证文件服务在客户端或者服务端出现问题的时候能正常使用是非常重要的。HDFS 的容错能力大概可以分为两个方面：文件系统的容错性以及 Hadoop 本身的容错能力。文件系统的容错性主要借助于以下几个手段：

（1）在目录节点和数据节点之间维持心跳检测。当由于网络故障之类的原因，导致数据节点（DataNode）发出的心跳包没有被目录节点（NameNode）正常收到的时候，目录节点就不会将任何新的 IO 操作派发给那个数据节点，该数据节点上的数据被认为是无效的，因此，目录节点会检测是否有文件块的副本数目小于设置值，如果小于就自动开始复制新的副本，并分发到其他数据节点上。

（2）检测文件块的完整性。HDFS 会记录每个新创建的文件的所有块的校验和。当以后检索这些文件的时候，从某个节点获取块，会首先确认校验和是否一致，如果不一致，会从其他数据节点上获取该块的副本。

（3）集群的负载均衡。由于节点的失效或者增加，可能导致数据分布的不均匀，当某个数据节点的空闲空间大于一个临界值的时候，HDFS 会自动从其他数据节点迁移数据过来。

（4）维护多个 FsImage 和 Editlog 的拷贝。目录节点上的 fsimage 和 edits 日志文件是 HDFS 的核心数据结构，如果这些文件损坏了，HDFS 将失效。因而，目录节点可以配置成

支持维护多个 FsImage 和 Editlog 的拷贝。任何对 FsImage 或者 Editlog 的修改，都将同步到它们的副本上。它总是选取最近的一致的 FsImage 和 Editlog 来使用。 目录节点在 HDFS 是单点存在的，如果目录节点所在的机器错误，手工的干预是必须的。

（5)文件的删除。一个文件被删除时，并不是马上从目录节点移除命名空间，而是放在 / trash 目录下，随时可恢复，直到超过设置时间才被正式移除。

再说 Hadoop 本身的容错性，Hadoop 支持升级和回滚，当升级 Hadoop 软件时出现 bug 或者不兼容现象，可以通过回滚恢复到老的 Hadoop 版本。

- **安全性问题**

HDFS 的安全性是比较弱的，只有简单的与 unix 文件系统类似的文件许可控制，未来版本会实现类似 NFS 的 Kerberos 验证系统。

本章小结

本章介绍 Hadoop 分布式文件系统 HDFS，介绍了块、目录节点、数据节点等相关概念，并介绍了 HDFS 体系结构、命名空间、存储原理、通讯协议、数据错误和异常、尚未实现的功能总结，最后讲述了从 HDFS 看分布式文件系统的设计需求。

总体而言，HDFS 作为通用的分布式文件系统并不适合，它在并发控制、缓存一致性以及小文件读写的效率上是比较弱的。但是它有自己明确的设计目标，那就是支持大的数据文件（TB 级），并且这些文件以顺序读为主，以文件读的高吞吐量为目标，并且与 MapReduce 框架紧密结合。

参考文献

[1] HDFS 详解. http://www.cnblogs.com/chinacloud/archive/2010/12/03/1895369.html

第 6 章 Zookeeper

Zookeeper 是 Hadoop 的一个子项目，是一种分布式的、开源的、应用于分布式应用的协作服务，主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。Zookeeper 很容易编程接入，它使用了一个和文件树结构相似的数据模型，可以使用 Java 或者 C 来进行编程接入。

在分布式应用中，由于工程师不能很好地使用锁机制，以及基于消息的协调机制不适合在某些应用中使用，因此，需要有一种可靠的、可扩展的、分布式的、可配置的协调机制来统一系统的状态。Zookeeper 的目的就在于此。

本章介绍 Zookeeper 的相关知识，内容要点如下：

- Zookeeper 简介
- Zookeeper 的工作原理
- Zookeeper 的数据模型
- Zookeeper 的典型应用场景

6.1 Zookeeper 简介

Zookeeper 是一种分布式的、开源的、应用于分布式应用的协作服务，它提供了一些简单的操作，可以实现统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。Zookeeper 很容易编程接入，它使用了一个和文件树结构相似的数据模型，可以使用 Java 或者 C 来进行编程接入。

众所周知，分布式的系统协作服务很难有让人满意的产品。这些协作服务产品很容易陷入一些诸如竞争选择条件或者死锁的陷阱中，而 Zookeeper 则可以很好地实现协作服务。

6.1.1 系统架构

Zookeeper 不仅可以单机提供服务，同时也支持多机组成集群来提供服务（如图 6-1 所示）。实际上 Zookeeper 还支持另外一种伪集群的方式，也就是可以在一台物理机上运行多个 Zookeeper 实例。

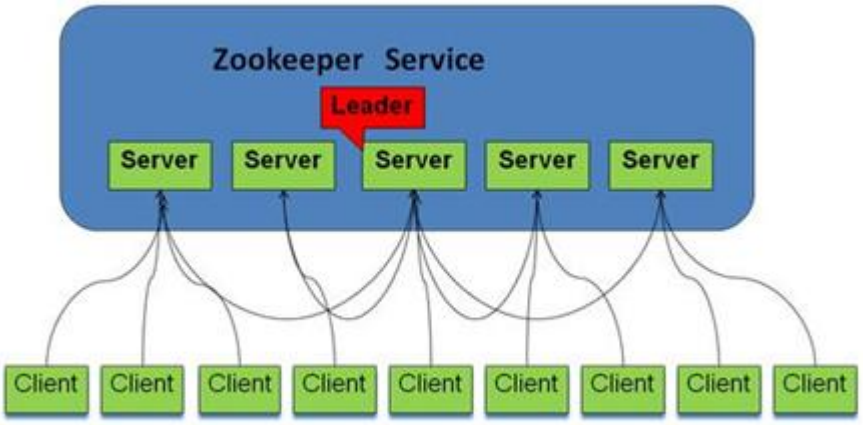

图 6-1 Zookeeper 的系统架构图

Zookeeper 中的角色主要包括领导者、学习者和客户端，如表 6-1 所示。

表 6-1 Zookeeper 中的角色

角色		描述
领导者（Leader）		领导者负责进行投票的发起和决议，更新系统状态
学习者（Learner）	跟随者（Follower）	Follower 用于接收客户请求并向客户端返回结果，在选主过程中参与投票
	观察者（ObServer）	ObServer 可以接收客户端连接，将写请求转发给 leader 节点。但 ObServer 不参加投票过程，只同步 leader 的状态。ObServer 的目的是为了扩展系统，提高读取速度
客户端（Client）		请求发起方

6.1.2 设计目的

Zookeeper 的设计目的包括以下几个方面：

- **最终一致性**：客户端不论连接到哪个服务器，展示给它的都是同一个视图，这是 Zookeeper 最重要的性能；
- **可靠性**：具有简单、健壮、良好的性能，如果消息被其中一台服务器接受，那么它将被所有的服务器接受；
- **实时性**：Zookeeper 保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息；但由于网络延时等原因，Zookeeper 不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用 sync()接口；

- **等待无关（wait-free）**：慢的或者失效的客户端不得干预快速的客户端的请求，使得每个客户端都能有效的等待；
- **原子性**：更新只能成功或者失败，没有中间状态；
- **顺序性**：包括全局有序和偏序两种：全局有序是指如果在一台服务器上消息 a 在消息 b 前发布，则在所有服务器上消息 a 都将在消息 b 前被发布；偏序是指如果一个消息 b 在消息 a 后被同一个发送者发布，a 必将排在 b 前面。

6.1.3 特点

Zookeeper 的特点主要包括以下几个方面：

- **Zookeeper 是简易的**

Zookeeper 通过一种和文件系统很像的层级命名空间来让分布式进程互相协同工作。这些命名空间由一系列数据寄存器组成，我们也叫这些数据寄存器为 znodes。这些 znodes 就有点像是文件系统中的文件和文件夹。和文件系统不一样的是，文件系统的文件是存储在存储区上的，而 Zookeeper 的数据是存储在内存上的。同时，这就意味着 Zookeeper 有着高吞吐和低延迟。

Zookeeper 实现了高性能、高可靠性和有序的访问。高性能保证了 Zookeeper 能应用在大型的分布式系统上。高可靠性保证它不会由于单一节点的故障而造成任何问题。有序的访问能保证客户端可以实现较为复杂的同步操作。

- **Zookeeper 是可用的**

组成 Zookeeper 的各个服务器必须要能相互通信。他们在内存中保存了服务器状态，也保存了操作的日志，并且持久化快照。只要大多数的服务器是可用的，那么 Zookeeper 就是可用的。

客户端连接到一个 Zookeeper 服务器，并且维持 TCP 连接，发送请求，获取回复，获取事件，并且发送连接信号。如果这个 TCP 连接断掉了，那么，客户端还可以连接另外一个服务器。

- **Zookeeper 是有序的**

Zookeeper 使用数字来对每一个更新进行标记，这样能保证 Zookeeper 交互的有序。后续的操作可以根据这个顺序实现诸如同步操作这样更高更抽象的服务。

- **Zookeeper 是高效的**

Zookeeper 的高效更表现在以读为主的系统上。Zookeeper 可以在上千台服务器组成的读写比例大约为 10:1 的分布系统上表现优异。

6.2 Zookeeper 的工作原理

Zookeeper 的核心是原子广播，这个机制保证了各个服务器之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，即恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式；当领导者被选举出来，且大多数 server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 server 具有相同的系统状态。

为了保证事务的顺序一致性，Zookeeper 采用了递增的事务 id 号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了 zxid。在实现时，zxid 是一个 64 位的数字，它的高 32 位是 epoch，用来表示 leader 关系是否改变，每次一个 leader 被选出来时，它都会有一个新的 epoch，表示当前属于那个 leader 的统治时期。低 32 位用于递增计数。

每个 server 在工作过程中有三种状态：

- LOOKING：当前 server 不知道 leader 是谁，正在搜寻；
- LEADING：当前 server 即为选举出来的 leader；
- FOLLOWING：leader 已经选举出来，当前 server 与之同步。

6.2.1 选主流程

当 leader 崩溃或者 leader 失去大多数的 follower，这时候 Zookeeper 就会进入恢复模式。恢复模式需要重新选举出一个新的 leader，让所有的 server 都恢复到一个正确的状态。Zookeeper 的选举算法有两种：一种是基于 basic paxos 实现的，另外一种是基于 fast paxos 算法实现的。系统默认的选举算法为 fast paxos。这里先介绍 basic paxos 流程：

- 选举线程由当前 Server 发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的 Server；
- 选举线程首先向所有 Server 发起一次询问(包括自己)；

- 选举线程收到回复后，验证是否是自己发起的询问(验证 zxid 是否一致)，然后获取对方的 id(myid)，并存储到当前询问对象列表中，最后获取对方提议的 leader 相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；
- 收到所有 Server 回复以后，就计算出 zxid 最大的那个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server；
- 选举线程将当前 zxid 最大的 Server 设置为当前 Server 要推荐的 Leader 后，如果此 Server 获得 n/2 + 1 的 Server 票数，则设置当前推荐的 leader 为获胜的 Server，产生选举结果 leader，然后，根据获胜的 Server 相关信息设置自己的状态；否则，继续这个过程，直到 leader 被选举出来。

通过流程分析我们可以得出：要使 leader 获得多数 Server 的支持，则 Server 总数必须是奇数 2n+1，且存活的 Server 的数目不得少于 n+1。

每个 Server 启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的 Server，还会从磁盘快照中恢复数据和会话信息，Zookeeper 会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。选主的具体流程图如图 6-2 所示。

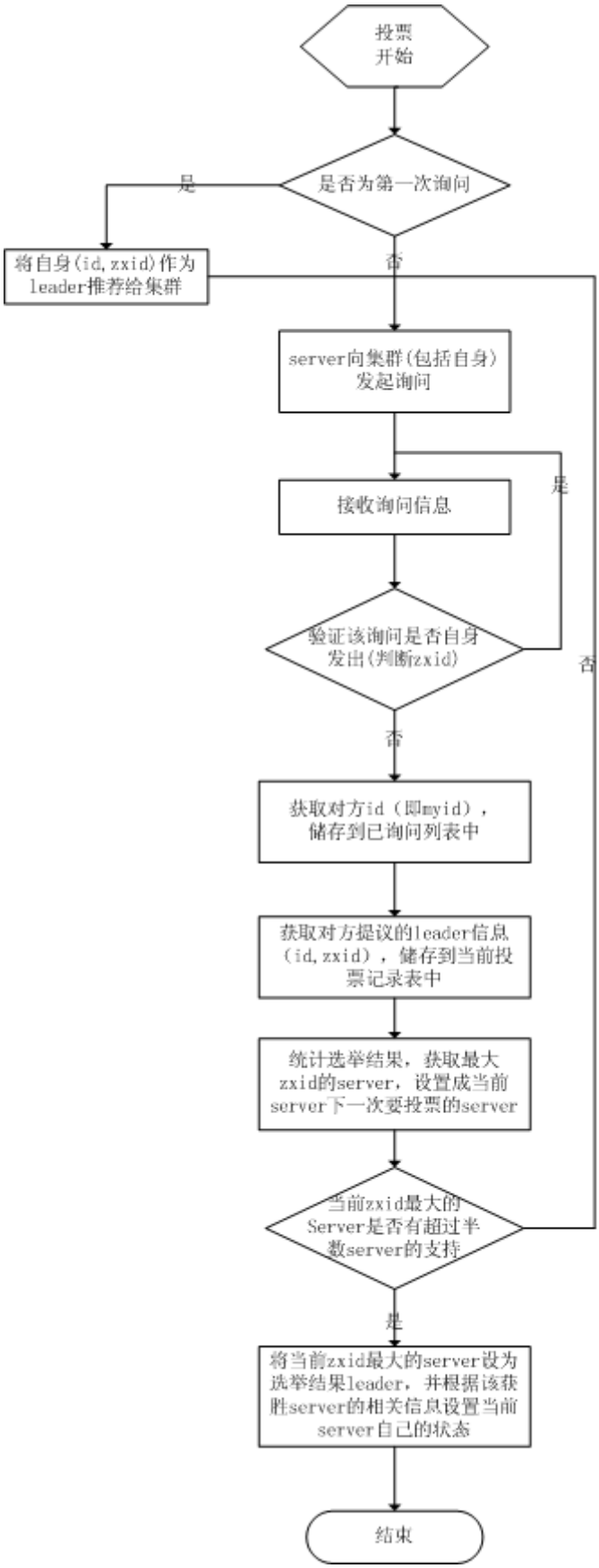

图 6-2 采用 basic paxos 算法实现选主流程

fast paxos 流程是在选举过程中，某 Server 首先向所有 Server 提议自己要成为 leader，当其它 Server 收到提议以后，解决 epoch 和 zxid 的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息。重复这个流程，最后一定能选举出 leader。其流程图如图 6-3 所示。

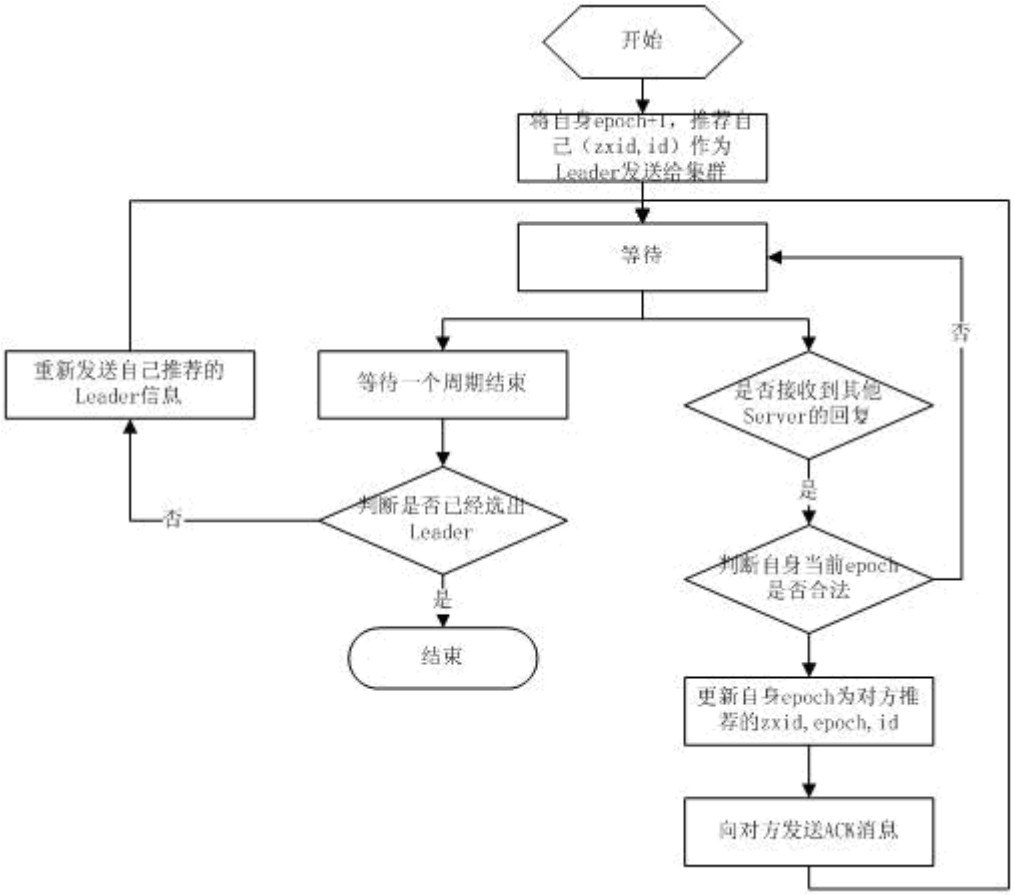

图 6-3 采用 fast paxos 算法实现选主流程

6.2.2 同步流程

选完 leader 以后，Zookeeper 就进入状态同步过程，具体如下：

- leader 等待 server 连接；
- Follower 连接 leader，将最大的 zxid 发送给 leader；
- Leader 根据 follower 的 zxid 确定同步点；
- 完成同步后通知 follower 已经成为 uptodate 状态；
- Follower 收到 uptodate 消息后，又可以重新接受 client 的请求进行服务了。

流程图如图 6-4 所示。

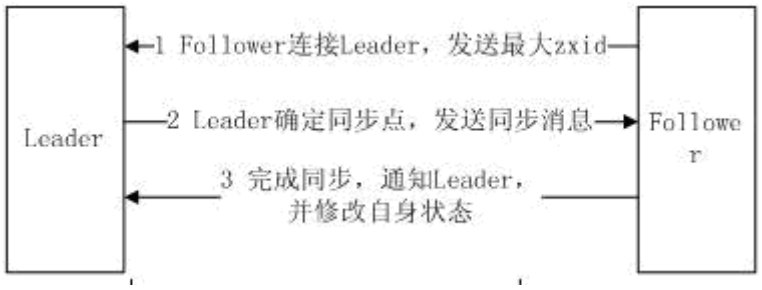

图 6-4 同步过程

6.2.3 工作流程

6.2.3.1 Leader 工作流程

Leader 主要有三个功能：

- 恢复数据；
- 维持与 Learner 的心跳，接收 Learner 请求并判断 Learner 的请求消息类型；
- Learner 的消息类型主要有 PING 消息、REQUEST 消息、ACK 消息、REVALIDATE 消息，根据不同的消息类型，进行不同的处理。

PING 消息是指 Learner 的心跳信息；REQUEST 消息是 Follower 发送的提议信息，包括写请求及同步请求；ACK 消息是 Follower 的对提议的回复，超过半数的 Follower 通过，则 commit 该提议；REVALIDATE 消息是用来延长 SESSION 有效时间。

Leader 的工作流程简图如图 6-5 所示，在实际实现中，流程要比该图复杂得多，启动了三个线程来实现功能。

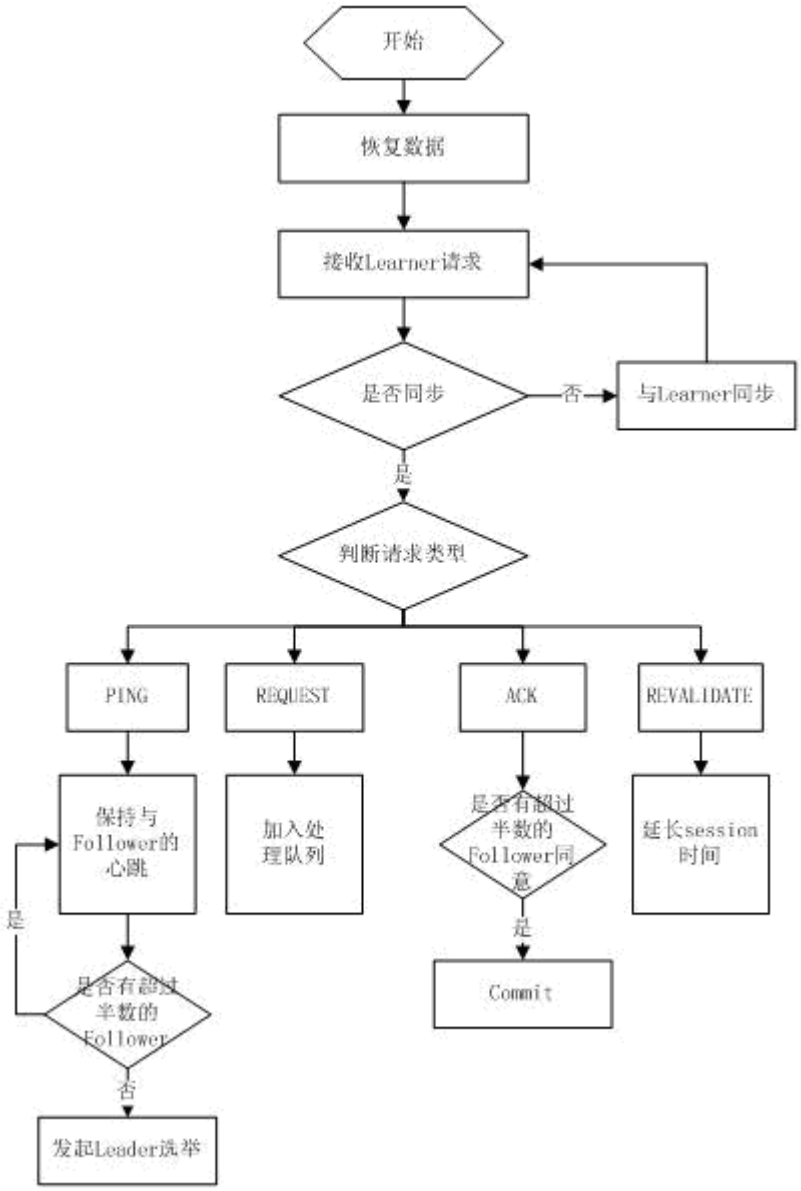

图 6-5 Leader 的工作流程

6.2.3.2 Follower 工作流程

Follower 主要有四个功能：

- 向 Leader 发送请求（PING 消息、REQUEST 消息、ACK 消息、REVALIDATE 消息）；
- 接收 Leader 消息并进行处理；
- 接收 Client 的请求，如果为写请求，发送给 Leader 进行投票；
- 返回 Client 结果。

Follower 循环处理如下几种来自 Leader 的消息：

- **PING** 消息： 心跳消息；
- **PROPOSAL** 消息：Leader 发起的提案，要求 Follower 投票；
- **COMMIT** 消息：服务器端最新一次提案的信息；
- **UPTODATE** 消息：表明同步完成；
- **REVALIDATE** 消息：根据 Leader 的 REVALIDATE 结果，来决定关闭待 revalidate 的 session 还是允许其接受消息；
- **SYNC** 消息：返回 SYNC 结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

Follower 的工作流程简图如图 6-6 所示，在实际实现中，Follower 是通过 5 个线程来实现功能的。

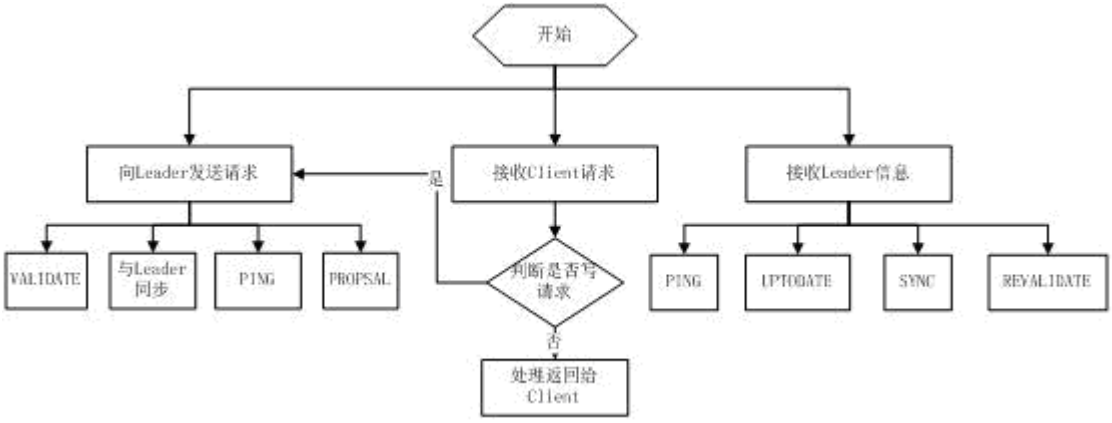

图 6-6 Follower 的工作流程

对于 observer 的流程不再叙述，observer 流程和 Follower 的唯一不同的地方就是 observer 不会参加 leader 发起的投票。

6.3 Zookeeper 的数据模型

Zookeeper 维护一个类似文件系统的数据结构，如图 6-7 所示。

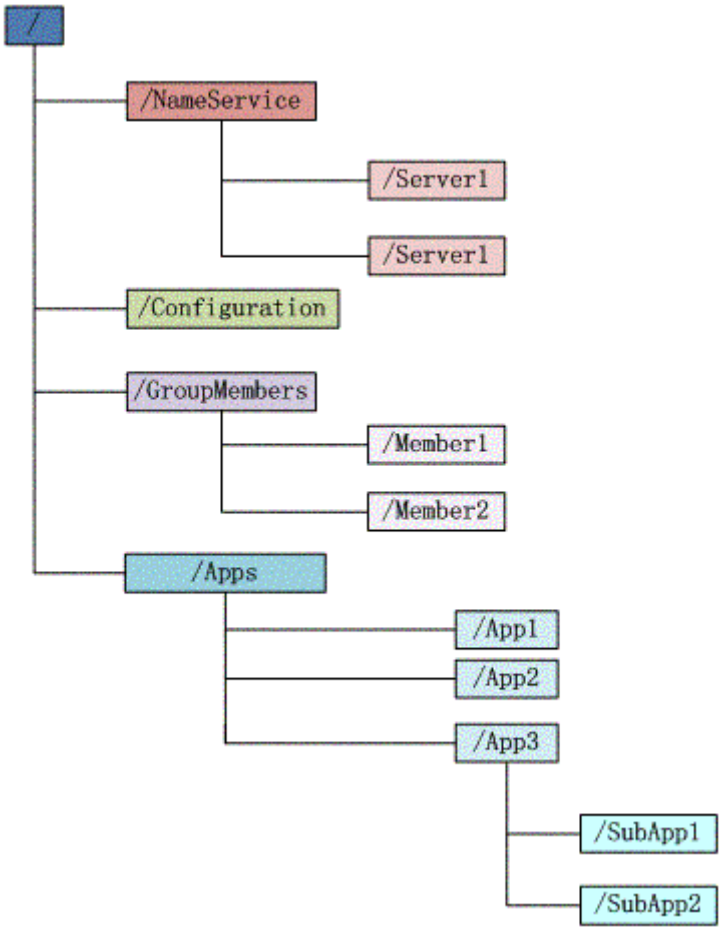

图 6-7 Zookeeper 的数据模型

每个子目录项如 NameService 都被称作为 znode，和文件系统一样，我们能够自由地增加、删除 znode，在一个 znode 下增加、删除子 znode，唯一的不同在于 znode 是可以存储数据的。Zookeeper 有四种类型的 znode：

- PERSISTENT-持久化目录节点：客户端与 Zookeeper 断开连接后，该节点依旧存在；
- PERSISTENT_SEQUENTIAL-持久化顺序编号目录节点：客户端与 Zookeeper 断开连接后，该节点依旧存在，只是 Zookeeper 会给该节点名称进行顺序编号；
- EPHEMERAL-临时目录节点：客户端与 Zookeeper 断开连接后，该节点被删除；
- EPHEMERAL_SEQUENTIAL-临时顺序编号目录节点：客户端与 Zookeeper 断开连接后，该节点被删除，只是 Zookeeper 会给该节点名称进行顺序编号。

Zookeeper 这种数据结构有如下这些特点：

- 每个子目录项如 NameService 都被称作为 znode，这个 znode 是被它所在的路径唯一标识的，如 Server1 这个 znode 的标识为/NameService/Server1；
- znode 可以有子节点目录，并且每个 znode 可以存储数据，注意 EPHEMERAL 类型的目录节点不能有子节点目录；
- znode 是有版本的，每个 znode 中存储的数据可以有多个版本，也就是一个访问路径中可以存储多份数据；
- znode 可以是临时节点，一旦创建这个 znode 的客户端与服务器失去联系，这个 znode 也将自动删除，Zookeeper 的客户端和服务器通信采用长连接方式，每个客户端和服务器通过心跳来保持连接，这个连接状态称为 session，如果 znode 是临时节点，这个 session 失效，znode 也就删除了；
- znode 的目录名可以自动编号，如 App1 已经存在，再创建的话，将会自动命名为 App2；
- znode 可以被监控，包括这个目录节点中存储的数据的修改，子节点目录的变化等，一旦发生变化就可以通知那些设置了监控该 znode 的客户端，这个是 Zookeeper 的核心特性，Zookeeper 的很多功能都是基于这个特性实现的，后面在典型的应用场景中会有实例介绍。

6.4 Zookeeper 的典型应用场景

6.4.1 统一命名服务

分布式应用中，通常需要有一套完整的命名规则，既能够产生唯一的名称又便于人识别和记住，通常情况下用树形的名称结构是一个理想的选择，树形的名称结构是一个有层次的目录结构，既对人友好又不会重复。说到这里你可能想到了 JNDI，没错，Zookeeper 的命名服务（Name Service）与 JNDI 能够完成的功能是差不多的，它们都是将有层次的目录结构关联到一定资源上，但是，Zookeeper 的 Name Service 更加是广泛意义上的关联，也许你并不需要将名称关联到特定资源上，你可能只需要一个不会重复名称，就像数据库中产生一个唯一的数字主键一样。

Name Service 已经是 Zookeeper 内置的功能，你只要调用 Zookeeper 的 API 就能实现。如调用 create 接口就可以很容易创建一个目录节点。

6.4.2 配置管理

配置的管理在分布式应用环境中很常见，例如，同一个应用系统需要多台 PC 服务器运行，但是，它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的配置项，那么，就必须同时修改每台运行这个应用系统的 PC 服务器，这样非常麻烦，而且容易出错。

像这样的配置信息完全可以交给 Zookeeper 来管理，将配置信息保存在 Zookeeper 的某个目录节点中，然后，将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。

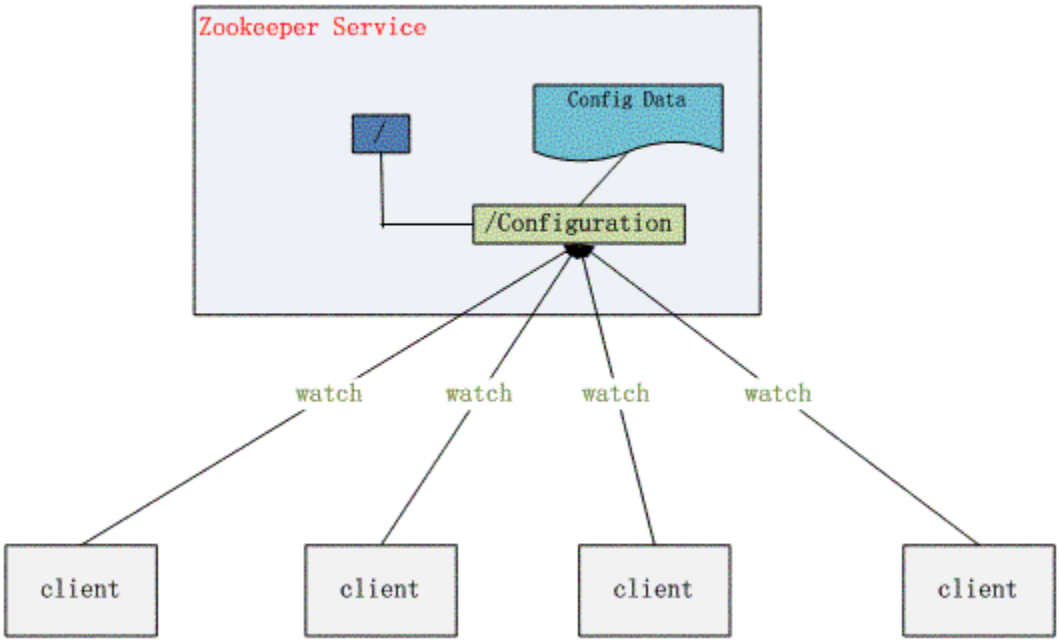

图 6-8 配置管理结构图

6.4.3 集群管理

Zookeeper 能够很容易地实现集群管理的功能，如果有多台 Server 组成一个服务集群，那么必须要一个“总管”知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，集群中其它机器必须知道，从而做出调整重新分配服务策略。同样，当增加集群的服务能力时，就会增加一台或多台 Server，同样也必须让“总管”知道。

Zookeeper 不仅能够帮你维护当前的集群中机器的服务状态，而且能够帮你选出一个“总管”，让这个总管来管理集群，这就是 Zookeeper 的另一个功能——Leader Election。

它们的实现方式都是在 Zookeeper 上创建一个 EPHEMERAL 类型的目录节点，然后每个 Server 在它们创建目录节点的父目录节点上调用 getChildren(String path, boolean watch)方法并设置 watch 为 true，由于是 EPHEMERAL 目录节点，当创建它的 Server 死去，这个目录节点也随之被删除，所以 Children 将会变化，这时 getChildren 上的 Watch 将会被调用，所以其它 Server 就知道已经有某台 Server 死去了。新增 Server 也是同样的原理。

Zookeeper 如何实现 Leader Election，也就是选出一个 Master Server 呢？和前面的一样，每台 Server 创建一个 EPHEMERAL 目录节点，不同的是，它还是一个 SEQUENTIAL 目录节点，所以它是个 EPHEMERAL_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号，我们可以选择当前是最小编号的 Server 为 Master，假如这个最小编号的 Server 死去，由于是 EPHEMERAL 节点，死去的 Server 对应的节点也被删除，所以当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免了传统意义上单 Master 容易出现单点故障的问题。

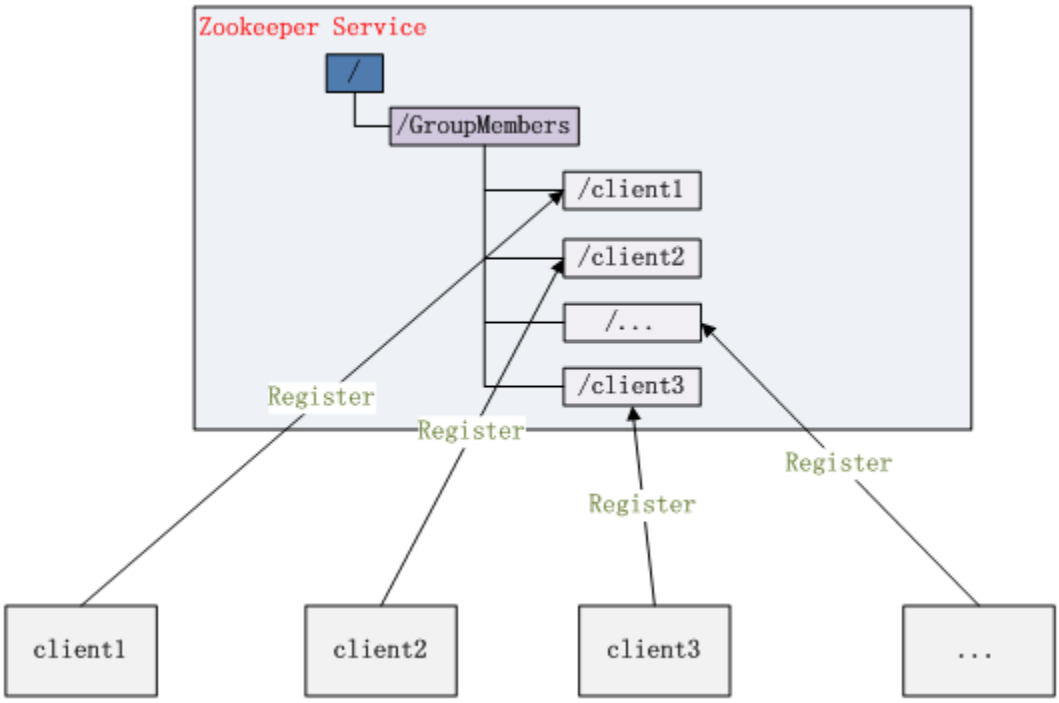

图 6-9 集群管理结构图

6.4.4 共享锁

共享锁在同一个进程中很容易实现，但是，在跨进程或者在不同 Server 之间就不好实现了。Zookeeper 却很容易实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL_SEQUENTIAL 目录节点，然后调用 getChildren 方法获取当前的目录节点列表中最小的目录节点是不是就是自己创建的目录节点，如果正是自己创建的，那么它就获得了这个锁，如果不是，那么它就调用 exists(String path, boolean watch)方法并监控 Zookeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，从而获得锁，释放锁很简单，只要删除前面它自己所创建的目录节点就行了。

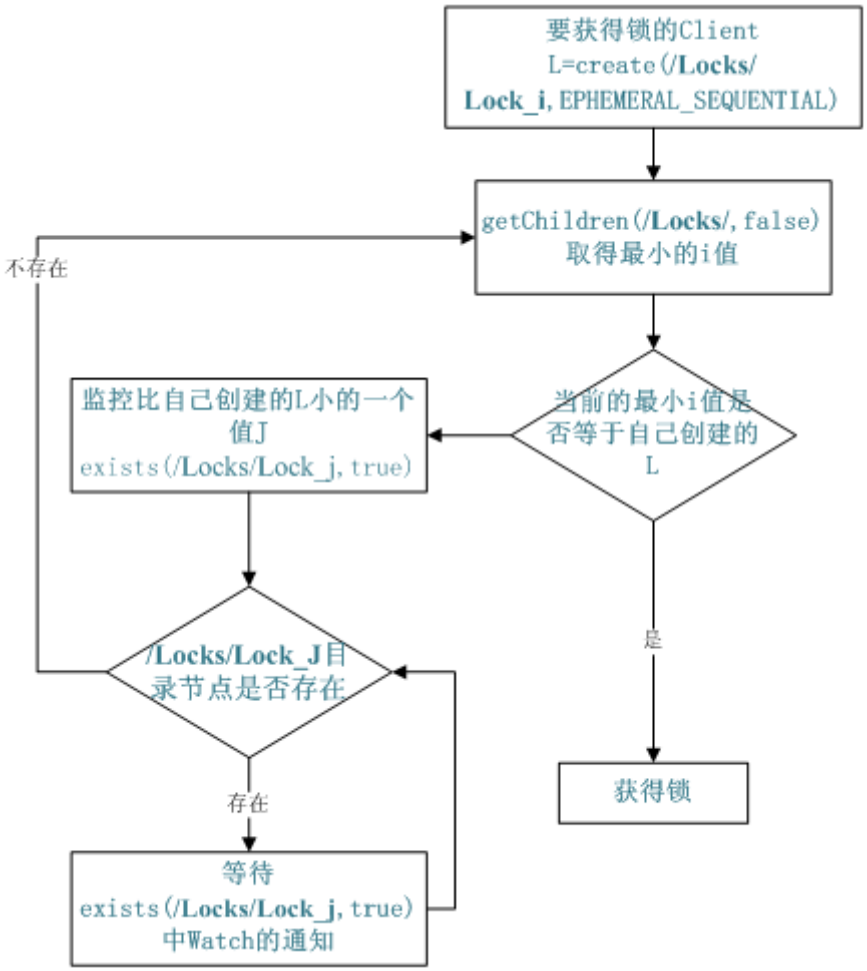

图 6-10 Zookeeper 实现 Locks 的流程图

6.4.5 队列管理

Zookeeper 可以处理两种类型的队列：

- 当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种是同步队列。
- 队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

同步队列用 Zookeeper 实现的实现思路如下：

创建一个父目录/synchronizing，每个成员都监控标志（Set Watch）位目录/synchronizing/start 是否存在，然后每个成员都加入这个队列，加入队列的方式就是创建/synchronizing/member_i 的临时目录节点，然后每个成员获取/ synchronizing 目录的所有目录节点，也就是 member_i。判断 i 的值是否已经是成员的个数，如果小于成员个数，就等待/synchronizing/start 的出现，如果已经相等就创建/synchronizing/start。

用下面的流程图更容易理解：

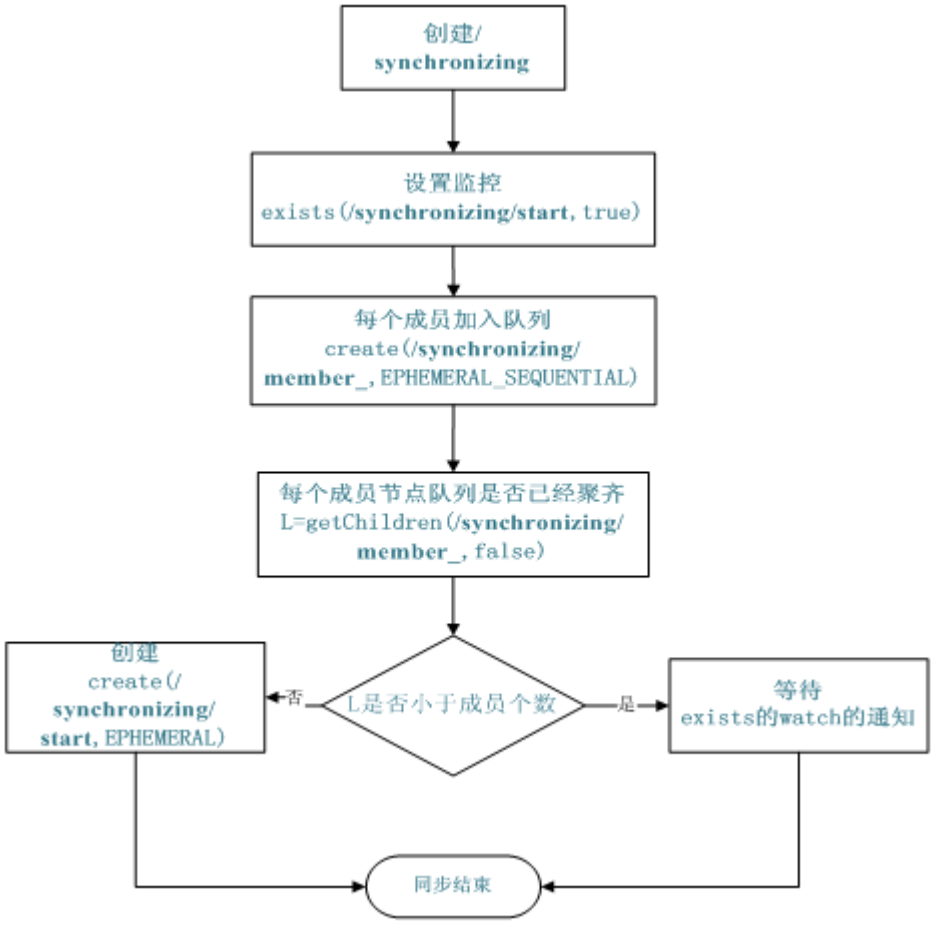

图 6-11 同步队列流程图

本章小结

Zookeeper 作为 Hadoop 项目中的一个子项目，是 Hadoop 集群管理的一个必不可少的模块，它主要用来控制集群中的数据，比如使用它来管理 Hadoop 集群中的 NameNode，还有 HBase 中 Master Election、Server 之间状态同步等。

本章介绍了 Zookeeper 的基本知识，并描述了几个典型的应用场景。这些都是 Zookeeper 的基本功能，最重要的是 Zookeeper 提供了一套很好的分布式集群管理的机制，就是它这种基于层次型的目录树的数据结构，并对树中的节点进行有效管理，从而可以设计出多种多样的分布式的数据管理模型，而不仅仅局限于上面提到的几个常用应用场景。

参考文献

[1] 分布式服务框架 Zookeeper -- 管理分布式环境中的数据. http://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/

[2] Zookeeper 与 paxos 算法. http://blog.csdn.net/ronghao100/article/details/7384752

第 7 章 HBase

HBase 是一个分布式的、面向列的开源数据库，该技术来源于 Google 公司发表的论文“BigTable：一个结构化数据的分布式存储系统”。HBase 是 Apache 的 Hadoop 项目的一个子项目。就像 BigTable 利用了 Google 文件系统 GFS（Google File System）所提供的分布式数据存储一样，HBase 在 Hadoop 文件系统 HDFS（Hadoop Distributed File System）之上提供了类似于 BigTable 的能力。HBase 不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库，而且 HBase 采用了基于列而不是基于行的模式。

本章介绍 HBase 相关知识，内容要点如下：

- HBase 简介
- HBase 使用场景和成功案例
- HBase 和传统关系数据库的对比分析
- HBase 访问接口

- HBase 数据模型
- HBase 系统架构
- HBase 存储格式
- 读写数据
- MapReduce on HBase

7.1HBase 简介

HBase—— Hadoop Database，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用 HBase 技术可在廉价 PC 服务器上搭建起大规模结构化存储集群。

HBase 是 Google BigTable 的开源实现，模仿并提供了基于 Google 文件系统的 BigTable 数据库的所有功能。类似 Google BigTable 利用 GFS 作为其文件存储系统，HBase 利用 Hadoop HDFS 作为其文件存储系统；Google 运行 MapReduce 来处理 BigTable 中的海量数据，HBase 同样利用 Hadoop MapReduce 来处理 HBase 中的海量数据；Google BigTable 利用 Chubby 作为协同服务，HBase 利用 Zookeeper 作为协同服务。

HBase 仅能通过行键(row key)和行键的值域区间范围（range）来检索数据，并且仅支持单行事务(可通过 Hive 支持来实现多表连接等复杂操作)。HBase 主要用来存储非结构化和半结构化的松散数据。

HBase 可以直接使用本地文件系统或者 Hadoop 作为数据存储方式，不过为了提高数据可靠性和系统的健壮性，发挥 HBase 处理大数据量等功能，需要使用 Hadoop 作为文件系统。与 Hadoop 一样，HBase 目标主要依靠横向扩展，通过不断增加廉价的商用服务器来增加计算和存储能力。

HBase 的目标是处理非常庞大的表，可以用普通的计算机处理超过 10 亿行数据并且由数百万列元素组成的数据表。

HBase 中的表一般有这样的特点：

- **大**：一个表可以有上亿行，上百万列；
- **面向列**：面向列(族)的存储和权限控制，列(族)独立检索；
- **稀疏**：对于为空(null)的列，并不占用存储空间，因此，表可以设计得非常稀疏。

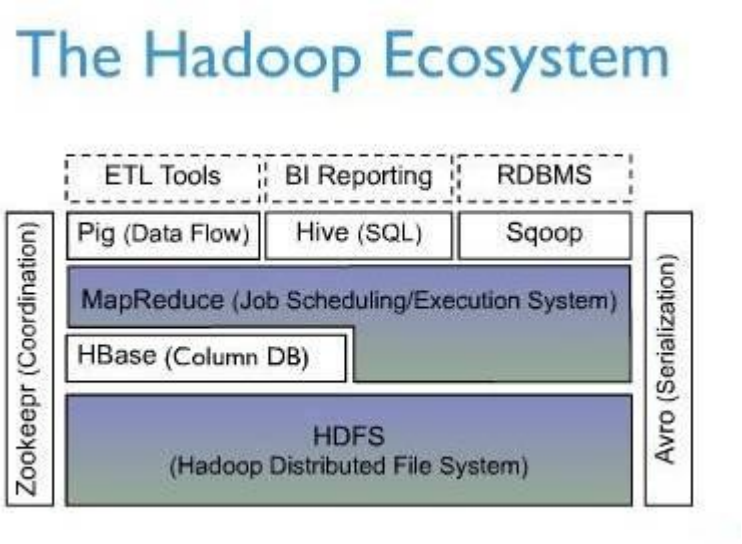

图 7-1 Hadoop 体系架构

图 7-1 描述了 Hadoop 生态系统中的各层系统，其中，HBase 位于结构化存储层，HDFS 为 HBase 提供了高可靠性的底层存储支持，MapReduce 为 HBase 提供了高性能的计算能力，Zookeeper 为 HBase 提供了稳定服务和失败恢复机制。

此外，Pig 和 Hive 还为 HBase 提供了高层语言支持，使得在 HBase 上进行数据统计处理变得非常简单。Sqoop 则为 HBase 提供了方便的 RDBMS 数据导入功能，使得传统数据库数据向 HBase 中迁移变得非常方便。

7.2HBase 使用场景和成功案例

HBase 被证实是一个强大的工具，尤其是在已经使用 Hadoop 的场合。在其“婴儿期”的时候，它就快速部署到了其他公司的生产环境，并得到开发人员的支持。今天，HBase 已经是 Apache 顶级项目，有着众多的开发人员和兴旺的用户社区。它成为一个核心的基础架构部件，运行在世界上许多公司（如 StumbleUpon、Trend Micro、Facebook、Twitter、Salesforce 和 Adobe）的大规模生产环境中。

HBase 不是数据管理问题的“万能药”，针对不同的使用场景你可能需要考虑其他的技术。让我们看看现在 HBase 是如何使用的，人们用它构建了什么类型的应用系统。通过这个讨论，你会知道哪种数据问题适合使用 HBase。

有时候了解软件产品的最好方法是看看它是怎么用的。它可以解决什么问题和这些解决方案如何适用于大型应用架构，这些能够告诉你很多。因为 HBase 有许多公开的产品部署案例，我们正好可以这么做。下面将详细介绍一些成功使用 HBase 的使用场景。

HBase 模仿了 Google 的 BigTable，让我们先从典型的 BigTable 问题开始：存储互联网。

7.2.1 典型的互联网搜索问题：BigTable 发明的原因

搜索是一种定位你所关心信息的行为。例如，搜索一本书的页码，其中含有你想读的主题，或者搜索网页，其中含有你想找的信息。搜索含有特定词语的文档，需要查找索引，该索引提供了特定词语和包含该词语的所有文档的映射。为了能够搜索，首先必须建立索引。Google 和其他搜索引擎正是这么做的。它们的文档库是整个互联网，搜索的特定词语就是你在搜索框里敲入的任何东西。

BigTable 和模仿出来的 HBase，为这种文档库提供存储，BigTable 提供行级访问，所以，爬虫可以在 BigTable 中插入和更新单个文档，每个文档保存为 BigTable 中的一行。MapReduce 计算作业运行在 BigTable 的整张表上，就可以高效地生成搜索索引，为网络搜索应用做准备。当用户发起网络搜索请求时，网络搜索应用就会查询已经建立好的索引，直接从 BigTable 中得到匹配的文档，然后把搜索结果提交给用户。图 7-2 显示了互联网搜索应用中 BigTable 的关键角色。

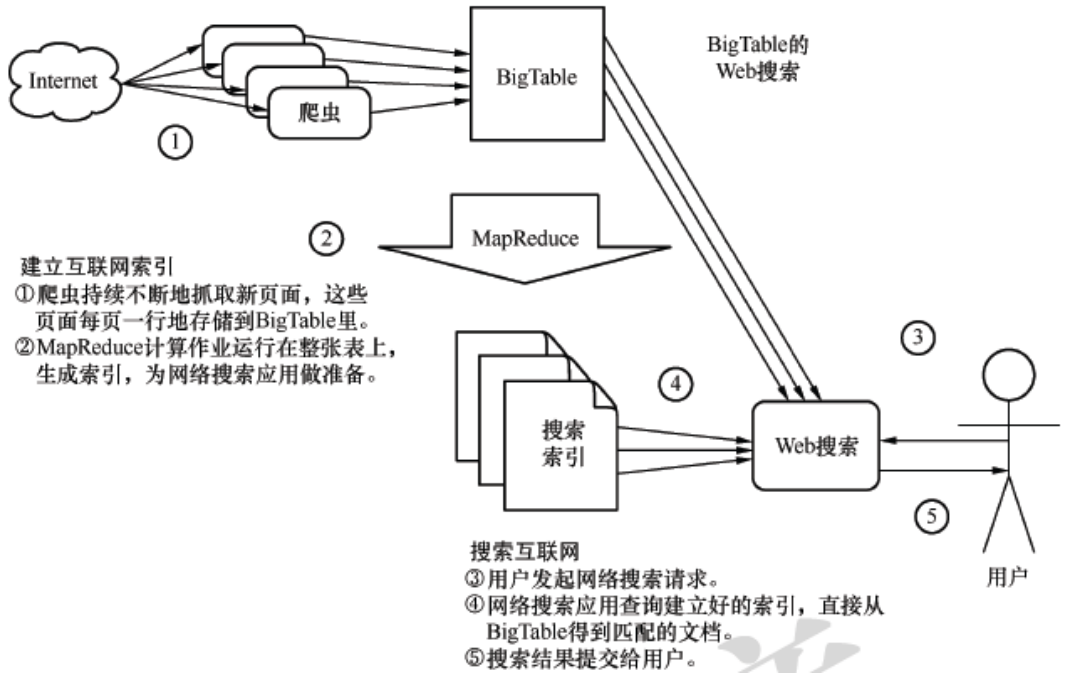

图 7-2 使用 BigTable 提供网络搜索结果

讲完典型 HBase 使用场景以后，我们来看看其他使用 HBase 的地方。愿意使用 HBase 的用户数量在过去几年里迅猛增长，部分原因在于 HBase 产品变得更加可靠且性能变得更好，更多原因在于越来越多的公司开始投入大量资源来支持和使用它。随着越来越多的商业

服务供应商提供支持，用户越发自信地把 HBase 应用于关键应用系统中。一个设计初衷是用 HBase 来存储互联网持续更新的网页副本，用在互联网相关的其他方面也是很合适的。例如，HBase 在社交网络公司内部和周围各种各样的需求中，也找到了用武之地。从存储个人之间的通信信息，到通信信息分析，HBase 成为了 Facebook、Twitter 和 StumbleUpon 等公司的关键基础设施。

在这个领域，HBase 有 3 种主要使用场景，但不限于这 3 种。接下来我们将介绍这 3 种主要的使用场景：（1）抓取增量数据；（2）内容服务；（3）信息交换。

7.2.2 抓取增量数据

数据通常是细水长流的，不断累加到已有的数据库中以备将来使用，如分析、处理和服务。许多 HBase 使用场景属于这一类——使用 HBase 作为数据存储，抓取来自各种数据源的增量数据。例如，这种数据源可能是网页爬虫（我们讨论过的 BigTable 典型问题），可能是记录用户看了什么广告和看了多长时间的广告效果数据，也可能是记录各种参数的时间序列数据。我们讨论几个成功的使用场景，以及这些项目涉及的公司。

1．抓取监控指标：OpenTSDB

服务数百万用户的、基于 Web 的产品的后台基础设施，一般都有数百或数千台服务器。这些服务器承担了各种功能——服务流量、抓取日志、存储数据和处理数据等等。为了保证产品正常运行，对服务器及其上面运行的软件的健康状态进行监控，是至关重要的（从 OS 到用户交互应用）。大规模监控整个环境，需要能够采集和存储来自不同数据源各种监控指标的监控系统。每个公司都有自己的办法。一些公司使用商业工具来收集和展示监控指标，而另外一些公司则采用开源框架。

StumbleUpon 创建了一个开源框架，用来收集服务器的各种监控指标。按照时间收集监控指标一般被称为时间序列数据，也就是说，按照时间顺序收集和记录的数据。StumbleUpon 的开源框架叫做 OpenTSDB，它是 Open Time Series Database（开放时间序列数据库）的缩写。这个框架使用 HBase 作为核心平台来存储和检索所收集的监控指标。创建这个框架的目的是为了拥有一个可扩展的监控数据收集系统，一方面，能够存储和检索监控指标数据并保存很长时间，另一方面，如果需要增加功能，也可以添加各种新监控指标。StumbleUpon 使用 OpenTSDB 监控所有基础设施和软件，包括 HBase 集群自身。

2．抓取用户交互数据：Facebook 和 StumbleUpon

抓取监控指标是一种使用方式，还有一种使用方式是抓取用户交互数据。如何跟踪数百万用户在网站上的活动？怎么知道哪一个网站功能最受欢迎？怎样使得这一次网页浏览直接影响到下一次？例如，谁看了什么？某个按钮被点击了多少次？还记得 Facebook 和 Stumble 里的 Like 按钮和 StumbleUpon 里的+1 按钮吗？是不是听起来像是一个计数问题？每次用户喜欢一个特定主题，计数器就增加一次。

StumbleUpon 在开始阶段采用的是 MySQL，但是，随着网站服务越来越流行，这种技术选择遇到了问题。急剧增长的用户在线负载需求，远远超过了 MySQL 集群的能力，最终，StumbleUpon 选择使用 HBase 来替换这些集群。当时，HBase 产品不能直接提供必需的功能。StumbleUpon 在 HBase 上做了一些小的开发改动，后来将这些开发工作贡献回了项目社区。

FaceBook 使用 HBase 的计数器来计量人们喜欢特定网页的次数。内容原创人和网页主人可以得到近乎实时的、多少用户喜欢他们网页的数据信息。他们可以因此更敏捷地判断应该提供什么内容。Facebook 为此创建了一个叫 Facebook Insights 的系统，该系统需要一个可扩展的存储系统。公司考虑了很多种可能的选择，包括关系型数据库管理系统、内存数据库和 Cassandra 数据库，最后决定使用 HBase。基于 HBase，Facebook 可以很方便地横向扩展服务规模，给数百万用户提供服务，还可以继续沿用他们已有的运行大规模 HBase 集群的经验。该系统每天处理数百亿条事件，记录数百个监控指标。

3．遥测技术：Mozilia 和 Trend Micro

软件运行数据和软件质量数据，不像监控指标数据那么简单。例如，软件崩溃报告是有用的软件运行数据，经常用来探究软件质量和规划软件开发路线图。HBase 可以成功地用来捕获和存储用户计算机上生成的软件崩溃报告。

Mozilla 基金会负责 FireFox 网络浏览器和 Thunderbird 电子邮件客户端两个产品。这些工具安装在全世界数百万台计算机上，支持各种操作系统。当这些工具崩溃时，会以 Bug 报告的形式返回一个软件崩溃报告给 Mozilla。Mozilla 如何收集这些数据？收集后又是怎么使用的呢？实际情况是这样的，一个叫做 Socorro 的系统收集了这些报告，用来指导研发部门研制更稳定的产品。Socorro 系统的数据存储和分析建构在 HBase 上。

使用 HBase，基本分析可以用到比以前多得多的数据。这种分析用来指导 Mozilla 的开发人员，使其更为专注，研制出 Bug 最少的版本。

Trend Micro 为企业客户提供互联网安全和入侵管理服务。安全的重要环节是感知，日志收集和分析对于提供这种感知能力是至关重要的。Trend Micro 使用 HBase 来管理网络信誉数据库，该数据库需要行级更新和支持 MapReduce 批处理。有点像 Mozilla 的 Socorro 系统，HBase 也用来收集和分析日志活动，每天收集数十亿条记录。HBase 中灵活的数据模式允许数据结构出现变化，当分析流程重新调整时，Trend Micro 可以增加新属性。

4．广告效果和点击流

过去十来年，在线广告成为互联网产品的一个主要收入来源。先提供免费服务给用户，在用户使用服务的时侯投放广告给目标用户。这种精准投放需要针对用户交互数据做详细的捕获和分析，以便理解用户的特征。基于这种特征，选择并投放广告。精细的用户交互数据会带来更好的模型，进而导致更好的广告投放效果，并获得更多的收入。但是，这类数据有两个特点：它以连续流的形式出现，它很容易按用户划分。理想情况下，这种数据一旦产生就能够马上使用，用户特征模型可以没有延迟地持续优化，也就是说，以在线方式使用。

HBase 非常适合收集这种用户交互数据，HBase 已经成功地应用在这种场合，它可以存储第一手点击流和用户交互数据，然后用不同的处理方式（MapReduce 是其中一种）来处理数据（清理、丰富和使用数据）。在这类公司，你会发现很多 HBase 案例。

7.2.3 内容服务

传统数据库最主要的使用场合之一是为用户提供内容服务。各种各样的数据库支撑着提供各种内容服务的应用系统。多年来，这些应用一直在发展，因此，它们所依赖的数据库也在发展。用户希望使用和交互的内容种类越来越多。此外，由于互联网迅猛的增长以及终端设备更加迅猛的增长，对这些应用的接入方式提出了更高的要求。各种各样的终端设备带来了另一个挑战：不同的设备需要以不同的格式使用同样的内容。

上面说的是用户消费内容（user consuming content），另外一个完全不同的使用场景是用户生成内容（user generate content）。Twitter 帖子、Facebook 帖子、Instagram 图片和微博等都是这样的例子。

它们的相同之处是使用和生成了许多内容。大量用户通过应用系统来使用和生成内容，而这些应用系统需要 HBase 作为基础。

内容管理系统（Content Management System，CMS）可以集中管理一切，可以用来存储

内容和提供内容服务。但是，当用户越来越多，生成的内容越来越多的时候，就需要一个更具可扩展性的 CMS 解决方案。可扩展的 Lily CMS 使用 HBase 作为基础，加上其他开源框架，如 Solr，构成了一个完整的功能组合。

Salesforce 提供托管 CRM 产品，这个产品通过网络浏览器界面提交给用户使用，显示出丰富的关系型数据库功能。在 Google 发表 NoSQL 原型概念论文之前很长一段时间，在生产环境中使用的大型关键数据库最合理的选择就是商用关系型数据库管理系统。多年来，Salesforce 通过数据库分库和尖端性能优化手段的结合扩展了系统处理能力，达到每天处理数亿事务的能力。

当 Salesforce 把分布式数据库系统列入选择范围后，他们评测了所有 NoSQL 技术产品，最后决定部署 HBase。一致性的需求是这个决定的主要原因。BigTable 类型的系统是唯一的架构方式，结合了无缝水平扩展能力和行级强一致性能力。此外，Salesforce 已经在使用 Hadoop 完成大型离线批处理任务，他们可以继续沿用 Hadoop 上面积累的宝贵经验。

1．URL 短链接

最近一段时间 URL 短链接非常流行，许多类似产品破土而出。StumbleUpon 使用名字为 su.pr.的短链接产品，这个产品以 HBase 为基础。这个产品用来缩短 URL，存储大量的短链接以及和原始长链接的映射关系，HBase 帮助这个产品实现扩展能力。

2．用户模型服务

经 HBase 处理过的内容往往并不直接提交给用户使用，而是用来决定应该提交给用户什么内容。这种中间处理数据用来丰富用户的交互。

还记得前面提到的广告服务场景里的用户特征吗？用户特征（或者说模型）就是来自 HBase。这种模型多种多样，可以用于多种不同场景。例如，针对特定用户投放什么广告的决定，用户在电商网站购物时实时报价的决定，用户在搜索引擎检索时增加背景信息和关联内容，等等。很多这种使用案例可能不便于公开讨论，说多了我们就有麻烦了。

当用户在电商网站上发生交易时，Runa 用户模型服务可以用来实时报价。这种模型需要基于不断产生的新用户数据持续调优。

7.2.4 信息交换

各种社交网络破土而出，世界变得越来越小。社交网站的一个重要作用就是帮助人们进行互动。有时互动在群组内发生（小规模和大规模），有时互动在两个个人之间发生。想想看，数亿人通过社交网络进行对话的场面。单单和远处的人对话还不足以让人满意，人们还想看看和其他人对话的历史记录。让社交网络公司感到幸运的是，保存这些历史记录很廉价，大数据领域的创新可以帮助他们充分利用廉价的存储。

在这方面，Facebook 短信系统经常被公开讨论，它也可能极大地推动了 HBase 的发展。当你使用 Facebook 时，某个时候你可能会收到或者发送短信给你的朋友。Facebook 的这个特性完全依赖于 HBase。用户读写的所有短信都存储在 HBase 里。Facebook 短信系统要求：高的写吞吐量，极大的表，数据中心内的强一致性。除了短信系统之外，其他应用系统要求：高的读吞吐量，计数器吞吐量，自动分库。工程师们发现 HBase 是一个理想的解决方案，因为它支持所有这些特性，它拥有一个活跃的用户社区，Facebook 运营团队在 Hadoop 部署上有丰富经验等等。在“Hadoop goes realtime at Facebook”这篇文章里，Facebook 工程师解释了这个决定背后的逻辑并展示了他们使用 Hadoop 和 HBase 构建在线系统的经验。

Facebook 工程师在 HBaseCon 2012 大会上分享了一些有趣的数据。在这个平台上每天交换数十亿条短信，每天带来大约 750 亿次操作。尖峰时刻，Facebook 的 HBase 集群每秒发生 150 万次操作。从数据规模角度看，Facebook 的集群每月增加 250TB 的新数据，这可能是已知的最大的 HBase 部署，无论是服务器的数量还是服务器所承载的用户量。

上述一些示例，解释了 HBase 如何解决一些有趣的老问题和新问题。你可能注意到一个共同点：HBase 可以用来对相同数据进行在线服务和离线处理。这正是 HBase 的独到之处。

7.3HBase 和传统关系数据库的对比分析

HBase 与以前的关系数据库存在很大的区别，它是按照 BigTable 来开发的，是一个稀疏的、分布的、持续多维度的排序映射数组。

HBase 就是这样一个基于列模式的映射数据库，它只能表示很简单的“键-数据”的映射关系，它大大简化了传统的关系数据库。二者具体区别如下：

- **数据类型**：HBase 只有简单的字符串类型，所有类型都是交由用户自己处理，它只

保存字符串。而关系数据库有丰富的类型选择和存储方式。

- **数据操作**：HBase 操作只有很简单的插入、查询、删除、清空等，表和表之间是分离的，没有复杂的表和表之间的关系，所以，不能也没有必要实现表和表之间的关联等操作。而传统的关系数据通常有各种各样的函数、连接操作。
- **存储模式**：HBase 是基于列存储的，每个列族都有几个文件保存，不同列族的文件是分离的。传统的关系数据库是基于表格结构和行模式保存的。
- **数据维护**：HBase 的更新，确切地说，应该不叫更新，而是一个主键或者列对应的新的版本，而它旧有的版本仍然会保留，所以，它实际上是插入了新的数据，而不是传统关系数据库里面的替换修改。
- **可伸缩性**：HBase 和 BigTable 这类分布式数据库就是直接为了这个目的开发出来的，能够轻易地增加或者减少（在硬件错误的时候）硬件数量，而且对错误的兼容性较高。而传统的关系数据库通常需要增加中间层才能实现类似的功能。

当前的关系数据库基本都是从上世纪 70 年代发展而来的，它们基本都有以下的体系特点：

- 面向磁盘存储和索引结构；
- 多线程访问；
- 基于锁的同步访问机制；
- 基于日志记录的恢复机制。

而 BigTable 和 HBase 之类基于列模式的分布式数据库，更适应海量存储和互联网应用的需求，灵活的分布式架构可以使其利用廉价的硬件设备组建一个大的数据仓库。互联网应用是以字符为基础的，BigTable 和 HBase 就是针对这些应用而开发出来的数据库。由于其中的时间戳特性，BigTable 和 HBase 与生俱来就特别适合于开发 wiki、archive.org 之类的服务，而 HBase 直接就是作为一个搜索引擎的一部分被开发出来的。

7.4HBase 访问接口

- **Native Java API**：最常规和高效的访问方式，适合 Hadoop MapReduce 作业并行批处理 HBase 表数据；
- **HBase Shell**：HBase 的命令行工具，最简单的接口，适合 HBase 管理使用；
- **Thrift Gateway**：利用 Thrift 序列化技术，支持 C++，PHP，Python 等多种语言，

适合其他异构系统在线访问 HBase 表数据；

- **REST Gateway**：支持 REST 风格的 Http API 访问 HBase，解除了语言限制；
- **Pig**：可以使用 Pig Latin 流式编程语言来操作 HBase 中的数据，和 Hive 类似，最终也是编译成 MapReduce 作业来处理 HBase 表数据，适合做数据统计；
- **Hive**：可以使用类似 SQL 语言来访问 HBase。

7.5 HBase 数据模型

7.5.1 概述

HBase 是一个类似 BigTable 的分布式数据库，大部分特性和 BigTable 一样，是一个稀疏的、长期存储的（存在硬盘上）、多维度的、排序的映射表。这张表的索引是行关键字、列关键字和时间戳。每个值是一个不解释的字符数组，数据都是字符串，没有类型。

用户在表中存储数据（如表 7-1 所示），每一行都有一个可排序的主键和任意多的列。由于是稀疏存储的，所以，同一张表里面的每一行数据都可以有截然不同的列。

列名字的格式是“<family>:<label>”，都是由字符串组成，每一张表有一个 family 集合，这个集合是固定不变的，相当于表的结构，只能通过改变表结构来改变。但是，label 值相对于每一行来说都是可以改变的。

HBase 把同一个 family 里面的数据存储在同一个目录底下，而 HBase 的写操作是锁行的，每一行都是一个原子元素，都可以加锁。

所有数据库的更新都有一个时间戳标记，每个更新都是一个新的版本，而 HBase 会保留一定数量的版本，这个值是可以设定的。客户端可以选择获取距离某个时间最近的版本，或者一次获取所有版本。

表 7-1 HBase 数据实例

Row Key	Timestamp	Column Family	
		URI	Parser
r1	t3	url=http://www.taobao.com	title=天天特价
	t2	host=taobao.com	

	t1		
r2	t5	url=http://www.alibaba.com	content=每天…
	t4	host=alibaba.com	

7.5.2 数据模型相关概念

在 HBase 数据模型中，包括如下三个重要概念：

- 行键（**Row Key**）：HBase 表的主键，表中的记录按照行键排序；
- 时间戳（**Timestamp**）：每次数据操作对应的时间戳，可以看作是数据的版本号；
- 列族（**Column Family**）：表在水平方向有一个或者多个列族组成，一个列族中可以由任意多个列组成，即列族支持动态扩展，无需预先定义列的数量以及类型，所有列均以二进制格式存储，用户需要自行进行类型转换。

1. 行键

行键是用来检索记录的主键。访问 HBase 表中的行，只有三种方式：

- 通过单个行键访问
- 通过行键的区间范围
- 全表扫描

行键可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10-100bytes)，在 HBase 内部，行键保存为字节数组。存储时，数据按照行键的字典序(byte order)排序存储。设计键时，要充分考虑这个特性，将经常一起读取的行存储放到一起 (位置相关性) 。注意：字典序对 int 排序的结果是 1，10，100，11，12，13，14，15，16，17，18，19，2，20，21，…，9，91，92，93，94，95，96，97，98，99。要保持整形的自然序，行键必须用 0 作左填充。

行的一次读写是原子操作 (不论一次读写多少列)。这个设计决策能够使用户很容易地理解程序在对同一个行进行并发更新操作时的行为。

2. 列族

HBase 表中的每个列都归属于某个列族。列族是表的模式的一部分(而列不是表的模式的一部分)，必须在使用表之前定义。列名都以列族作为前缀。例如 courses:history，

courses:math 都属于 courses 这个列族。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。实际应用中，列族上的控制权限能帮助我们管理不同类型的应用：我们允许一些应用可以添加新的基本数据，一些应用可以读取基本数据并创建继承的列族，一些应用则只允许浏览数据（甚至可能因为隐私的原因不能浏览所有数据）。

3. TimeStamp

HBase 中通过行和列确定的一个存储单元称为 cell。每个 cell 都保存着同一份数据的多个版本。不同的版本是通过时间戳来进行索引的，时间戳的类型是 64 位整型。时间戳可以由 HBase (在数据写入时自动)赋值，此时，时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 cell 中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。cell 中的数据是没有类型的，全部是字节码形式存储。

为了避免数据存在过多版本造成的管理 (包括存储和索引)负担，HBase 提供了两种数据版本回收方式。一是保存数据的最后 n 个版本，二是保存最近一段时间内的版本（比如最近七天）。用户可以针对每个列族进行设置。

7.5.3 概念视图

一个表可以想象成一个大的映射关系，通过主键，或者主键+时间戳，可以定位一行数据，由于是稀疏数据，所以某些列可以是空白的，表 7-2 就是 HBase 存储数据的概念视图。

表 7-2 HBase 数据的概念视图

Row Key	**Time Stamp**	**Column** *"contents:"*	**Column** *"anchor:"*		**Column** *"mime:"*
"com.cnn.www"	t9		"anchor:cnnsi.com"	"CNN"	
	t8		"anchor:my.look.ca"	"CNN.com"	
	t6	"<html>..."			"text/html"
	t5	"<html>..."			
	t3	"<html>..."			

表 7-2 是一个存储 Web 网页的范例列表片断。行名是一个反向 URL（即 com.cnn.www）。contents 列族用来存放网页内容，anchor 列族存放引用该网页的锚链接文本。CNN 的主页被 Sports Illustrater（即所谓 SI，CNN 的王牌体育节目）和 MY-look 的主页引用，因此，该行包含了名叫“anchor:cnnsi.com”和 “anchhor:my.look.ca”的列。每个锚链接只有一个版本（由时间戳标识，如 t9，t8）；而 contents 列则有三个版本，分别由时间戳 t3，t5 和 t6 标识。

7.5.4 物理视图

虽然从概念视图来看，HBase 中的每个表格是由很多行组成的，但是，在物理存储上面，它是按照列来保存的，这点在数据设计和程序开发的时候必须牢记。表 7-2 的概念视图在物理存储的时候应该表现成类似表 7-3 的样子。

表 7-3 HBase 数据的物理视图

Row Key	**Time Stamp**	**Column** *"contents:"*
"com.cnn.www"	t6	"<html>..."
	t5	"<html>..."
	t3	"<html>..."

Row Key	**Time Stamp**	**Column** *"anchor:"*	
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"

Row Key	**Time Stamp**	**Column** *"mime:"*
"com.cnn.www"	t6	"text/html"

需要注意的是，在概念视图（见表 7-2）上面有些列是空白的，这样的列实际上并不会被存储，当请求这些空白的单元格的时候，会返回 null 值。

如果在查询的时候不提供时间戳，那么会返回距离现在最近的那一个版本的数据。因为

在存储的时候，数据会按照时间戳排序。

例子 1：一个程序写 10 行数据，row[0-9]，先写入 anchor:foo 列，再写入 anchor:bar 列，最后重复写入 anchor:foo 列，由于是同一个列族，写到同一个映射文件里面，最后写到文件里面是下面这个样子的：

```
row=row0， column=anchor:bar， timestamp=1174184619081
row=row0， column=anchor:foo， timestamp=1174184620720
row=row0， column=anchor:foo， timestamp=1174184617161
row=row1， column=anchor:bar， timestamp=1174184619081
row=row1， column=anchor:foo， timestamp=1174184620721
row=row1， column=anchor:foo， timestamp=1174184617167
row=row2， column=anchor:bar， timestamp=1174184619081
row=row2， column=anchor:foo， timestamp=1174184620724
row=row2， column=anchor:foo， timestamp=1174184617167
row=row3， column=anchor:bar， timestamp=1174184619081
row=row3， column=anchor:foo， timestamp=1174184620724
row=row3， column=anchor:foo， timestamp=1174184617168
row=row4， column=anchor:bar， timestamp=1174184619081
row=row4， column=anchor:foo， timestamp=1174184620724
row=row4， column=anchor:foo， timestamp=1174184617168
row=row5， column=anchor:bar， timestamp=1174184619082
row=row5， column=anchor:foo， timestamp=1174184620725
row=row5， column=anchor:foo， timestamp=1174184617168
row=row6， column=anchor:bar， timestamp=1174184619082
row=row6， column=anchor:foo， timestamp=1174184620725
row=row6， column=anchor:foo， timestamp=1174184617168
row=row7， column=anchor:bar， timestamp=1174184619082
row=row7， column=anchor:foo， timestamp=1174184620725
row=row7， column=anchor:foo， timestamp=1174184617168
```

```
row=row8， column=anchor:bar， timestamp=1174184619082
row=row8， column=anchor:foo， timestamp=1174184620725
row=row8， column=anchor:foo， timestamp=1174184617169
row=row9， column=anchor:bar， timestamp=1174184619083
row=row9， column=anchor:foo， timestamp=1174184620725
row=row9， column=anchor:foo， timestamp=1174184617169
```

其中 anchor:foo 被保存了两次，由于时间戳不同，是两个不同的版本，而最新的数据排在前面，所以最新那次更新会先被找到。

7.6HBase 的实现

7.6.1 表和 HRegion

HBase 实现包括三个主要的功能组件：(1)库函数：链接到每个客户端；(2)一个 HMaster 主服务器；(3) 许多个 HRegion 服务器。HRegion 服务器可以根据工作负载的变化，从一个簇中动态地增加或删除。主服务器 HMaster 负责把 HRegion（类似于 BigTable 中的 Tablet）分配到 HRegion 服务器，探测 HRegion 服务器的增加和过期，进行 HRegion 服务器的负载均衡，以及 HDFS 文件系统中的垃圾收集。除此以外，它还处理模式变化，比如表和列族的创建。

每个 HRegion 服务器管理一个 HRegion 集合，通常在每个 HRegion 服务器上，会放置 10 到 1000 个 HRegion。HRegion 服务器处理针对那些已经加载的 HRegion 而提出的读写请求，并且会对过大的 HRegion 进行划分。

就像许多单服务器分布式存储系统一样，客户端并不是直接从主服务器读取数据，而是直接从 HRegion 服务器上读取数据。因为 HBase 客户端并不依赖于主服务器 HMaster 来获得 HRegion 的位置信息，所以，大多数客户端从来不和主服务器 HMaster 通信，从而使得在实际应用中，主服务器负载很小。

一个 HBase 中存储了许多表。每个表都是一个 HRegion 集合，每个 HRegion 包含了位于某个值域区间内的所有数据。在最初阶段，每个表只包含一个 HRegion。随着表的增长，它会被自动分解成许多 HRegion，每个 HRegion 默认尺寸大约是 100 到 200MB。

注：本章内容是林子雨老师根据网络资料整理编写的，对于这部分内容，网络资料的描述并不规范。比如，网络资料中，某个地方正文文字描述用 HRegion，而图片中文字或其他地方的文字描述却用没有“H”前缀的 Region，类似的情形包括 HStore 和 Store，HRegionServer 和 RegionServer，HMaster 和 Master 等，其实二者是等价的，即 HRegion 和 Region 是同一个概念。为了规范，本章内容在正文描述部分，全部统一成 HRegion、HStore 和 HRegionServer 这种格式，但是，由于图片无法修改（重新画图工作量比较大，暂时没有时间做这个工作），因此，图片中可能仍然出现 Region、Store、RegionServer 等概念。

关于 HBase 中的表和 HRegion 的概念，总结如下：

- 表中的所有行都按照行键的字典序排列；
- 表在行的方向上分割为多个 HRegion（如图 7-3 所示）；

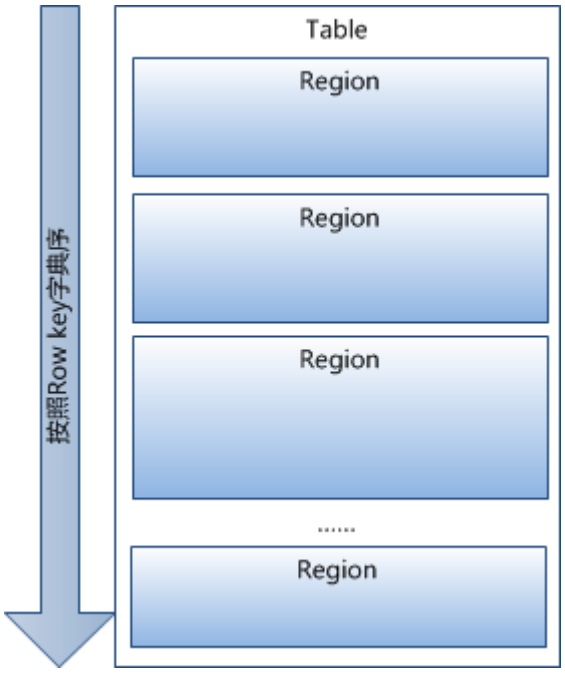

图 7-3 HBase 中的一个表包含多个 HRegion

- HRegion 会按照大小进行分割，每个表一开始只有一个 HRegion，随着数据不断插入表，HRegion 不断增大，当增大到一个阀值的时候，HRegion 就会等分成两个新的 HRegion（如图 7-4 所示）。表中的行不断增多，就会有越来越多的 HRegion。一个 HRegion 由[startkey，endkey)表示，不同的 HRegion 会被 HMaster 分配给相应的 HRegionServer 进行管理。

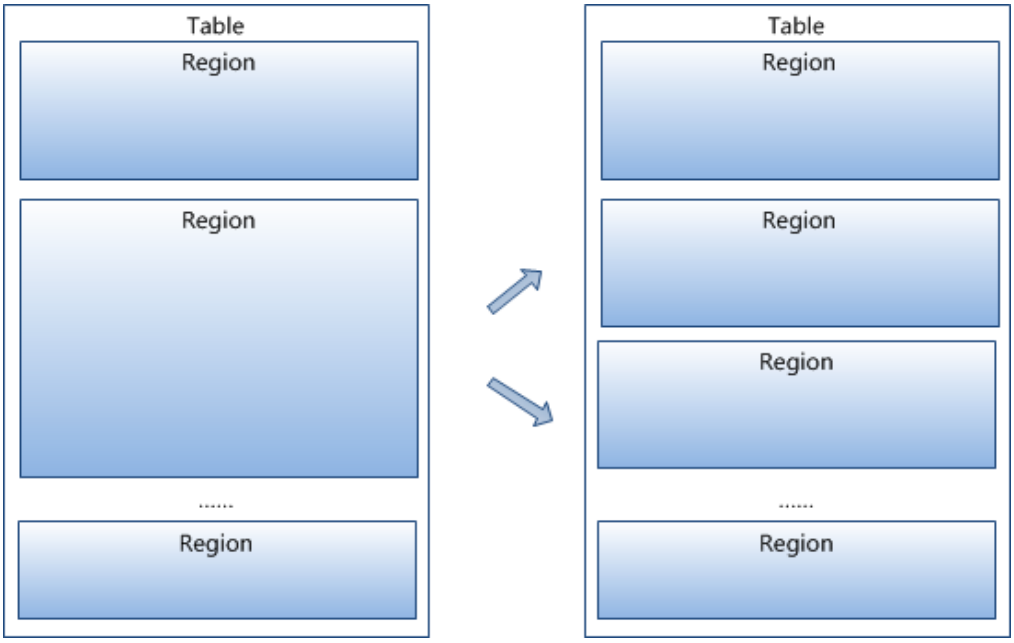

图 7-4 HBase 中的一个 HRegion 会分裂成多个新的 HRegion

- HRegion 是 HBase 中分布式存储和负载均衡的最小单元。最小单元就表示不同的 Hregion 可以分布在不同的 HRegionServer 上，但是，同一个 HRegion 是不会拆分到多个 HRegionServer 上的（如图 7-5 所示）。

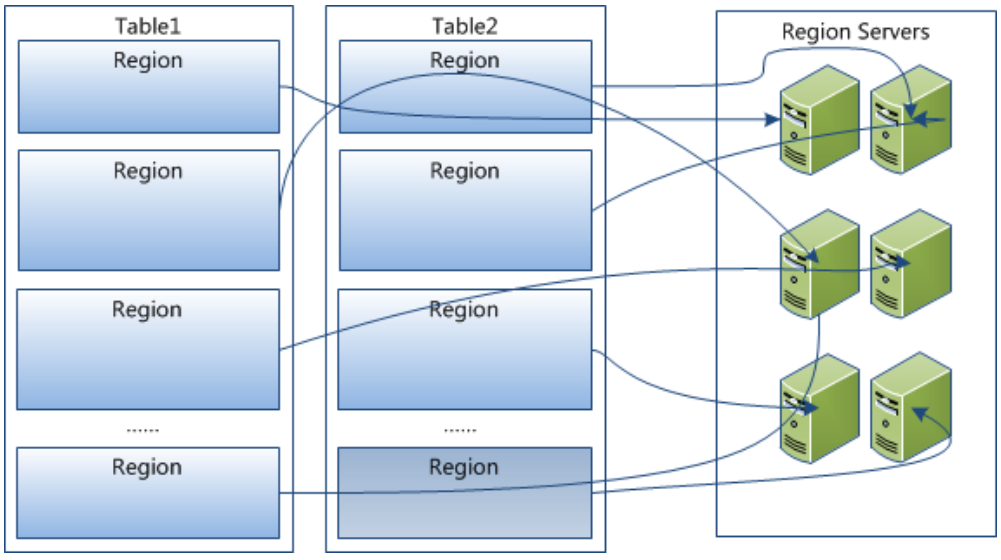

图 7-5 不同的 HRegion 可以分布在不同的 HRegionServer 上

- HRegion 虽然是分布式存储的最小单元，但并不是底层存储的最小单元。事实上，HRegion 由一个或者多个 HStore 组成，每个 HStore 保存一个列族。每个 HStrore 又由一个 memStore 和 0 至多个 HStoreFile 组成。HStoreFile 以 HFile 格式保存在 HDFS 上（如图 7-6 所示）。

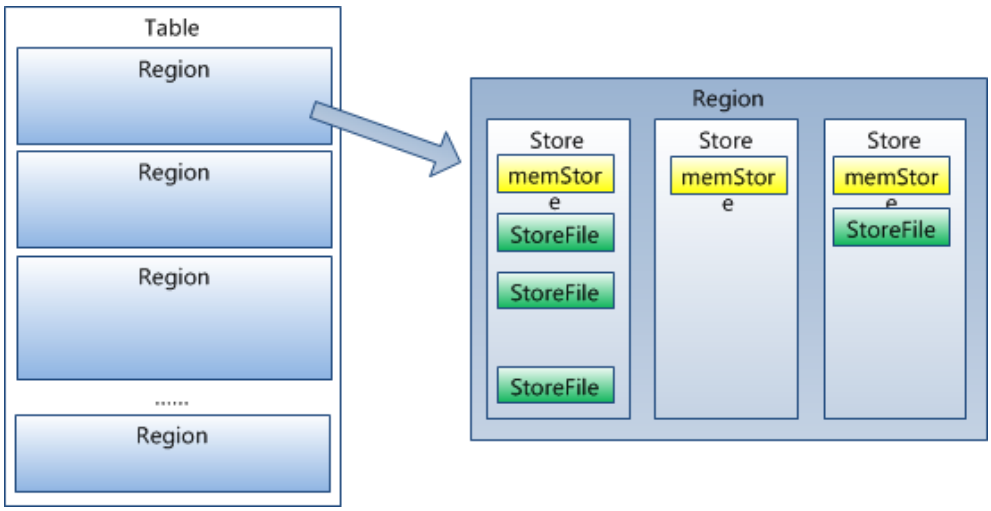

图 7-6 每个 HStore 由一个 memStore 和 0 至多个 HStoreFile 组成

7.6.2 HRegion 的定位

7.6.2.1 HBase 三层结构

HBase 使用类似 B+树的三层结构来保存 HRegion 位置信息：

- 第一层是 Zookeeper 文件：它记录了-ROOT-表的位置信息，即 root region 的位置信息；
- 第二层是-ROOT-表：只包含一个 root region，记录了.META.表中的 region 信息。通过 root region，我们就可以访问.META.表的数据。
- 第三层是.META.表：是一个特殊的表，记录了用户表的 HRegion 信息，.META.表可以有多个 HRegion，保存了 HBase 中所有数据表的 HRegion 位置信息。

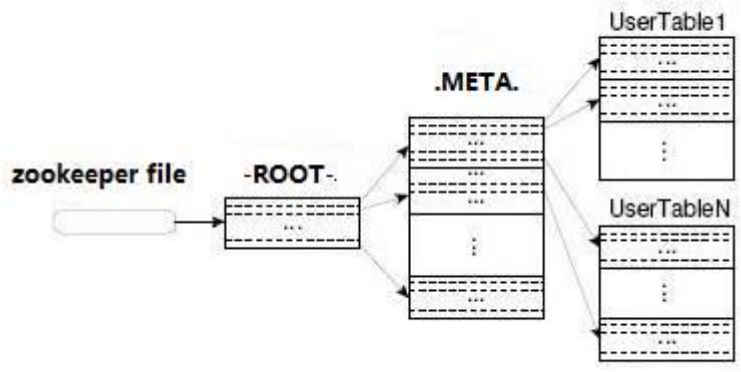

图 7-7 HBase 的三层结构

Client 访问用户数据之前，需要首先访问 Zookeeper，然后访问-ROOT-表，接着访问.META.表，最后才能找到用户数据的位置去访问，中间需要多次网络操作，不过 client 端会做 cache 缓存（如图 7-8 所示）。

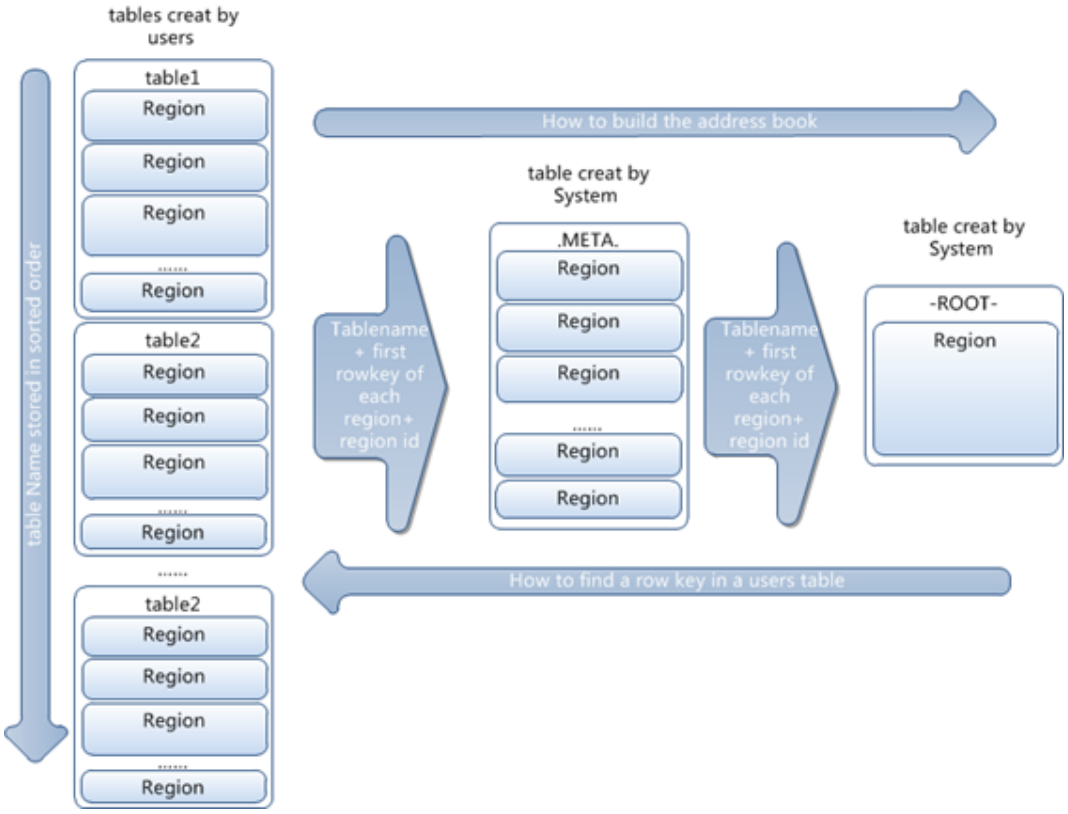

图 7-8 Client 访问用户数据过程

关于 HRegion 定位，需要进行几点说明：

- root region 永远不会被分裂，保证了最多需要三次跳转，就能定位到任意 HRegion；
- .META.表每行保存一个 HRegion 的位置信息，行键采用“表名+HRegion 的第一个行键+HRegion 的 ID”编码而成；
- 为了加快访问，.META.表的全部 HRegion 都保存在内存中；假设.META.表的一行在内存中大约占用 1KB，并且每个 HRegion 限制为 128MB。那么，上面的三层结构可以保存的 HRegion 数目为：(128MB/1KB) * (128MB/1KB) = 2^{34} 个 HRegion；
- client 会将查询过的位置信息保存缓存起来，缓存不会主动失效，因此，如果 client 上的缓存全部失效，则需要进行 6 次网络来回，才能定位到正确的 HRegion(其中三次用来发现缓存失效，另外三次用来获取位置信息)。

7.6.2.2 关于.META.表

之前我们说过，HRegion 是按照表名和主键范围区分的，由于主键范围是连续的，所以一般用开始主键就可以表达出来。

但是，如果只有开始主键还是不够的，因为我们有合并和分割操作，如果正好在执行这些操作的过程中出现死机，那么就可能存在多份表名和开始主键一样的数据，这个就要通过 HBase 的元数据信息来区分哪一份才是正确的数据文件了，为了区分这样的情况，每个 HRegion 都有一个 HRegionId 来标识它的唯一性。

所以，一个 HRegion 的表达式最后是：表名+开始主键+唯一 id（tablename + startkey + regionId）。

例子：hbaserepository，w-nk5YNZ8TBb2uWFIRJo7V==，6890601455914043877

我们可以用这个识别符来区分不同的 HRegion，这些数据就称为“元数据”。而元数据本身也是被保存在 HRegion 里面的，我们称呼这个表为“元数据表”(.META.表)，里面保存的就是 HRegion 标识符和实际 HRegion 服务器的映射关系。

元数据表本身也会增长，并且可能被分割为几个 HRegion，为了定位这些 HRegion，有一个根数据表（ROOT table），保存了所有元数据表的位置，而根数据表是不能被分割的，永远之存在一个 HRegion。

在 HBase 启动的时候，主服务器先去扫描根数据表，因为这个表只会有一个 HRegion，所以这个 HRegion 的名字是被写死的。当然要把根数据表分配到一个 HRegion 服务器需要一定的时间。

当根数据表被分配好之后，主服务器就会去扫描根数据表，获取元数据表的名字和位置，然后把元数据表分配到不同的 HRegion 服务器。

最后就是扫描元数据表，找到所有 HRegion 区域的信息，然后把它们分配给不同的 HRegion 服务器。

主服务器在内存中保存着当前活跃的 HRegion 服务器的数据，因此，如果主服务器死机的话，整个系统也就无法访问了，而服务器的信息也没有必要保存到文件里面。

元数据表和根数据表的每一行都包含一个列族——info 列族：

- info:regioninfo：包含了一个串行化的 HRegionInfo 对象。
- info:server：保存了一个字符串，是服务器地址 HServerAddress.toString()。
- info:startcode：一个长整型的数字的字符串，是 HRegion 服务器启动的时候传给主

服务器的，让主服务器决定这个 Hregion 服务器的信息有没有更改。

因此，当一个客户端从 Zookeeper 服务器上拿到根数据表地址以后，就可以直接访问相应的 HRegion 服务器获得数据，没有必要再连接主服务器。因此，主服务器的负载相对就小了很多，它只会处理超时的 HRegion 服务器，在启动的时候扫描根数据表和元数据表，和返回根数据表的 HRegion 服务器地址。

因此，HBase 的客户端是十分复杂的，它经常需要浏览元数据表和根数据表，在查询表的时候，如果一个 HRegion 服务器死机或者它上面的数据更改了，客户端就会继续重试，客户端保留的映射关系并不会一直正确的。这里的机制还需要进一步完善。

7.6.2.3 总结

关于 HRegion 的定位，总结如下：

- HRegion 服务器提供对 HRegion 的访问，一个 HRegion 只会保存在一个 HRegion 服务器上面。
- HRegion 会注册到主服务器上面。
- 如果主服务器死机，那么整个系统都会无效。
- 当前的 HRegion 服务器列表只有主服务器知道。
- HRegion 区域和 HRegion 服务器的对应关系保存在两个特别的 HRegion 里面（根数据表和元数据表），它们像其它 HRegion 一样被分配到不同的服务器。
- 根数据表是最特别的一个表，主服务器永远知道它的位置（在程序中写死）。
- 客户端需要自己浏览这些表，来找到数据在哪里。

7.7HBase 系统架构

图 7-9 给出了 HBase 的系统架构，围绕该架构，下面我们将分别介绍 Client、Zookeeper、HMaster、HRegionServer、HStore、HRegion 分配、HRegionServer 上线、HRegionServer 下线、HMaster 上线和 HMaster 下线。

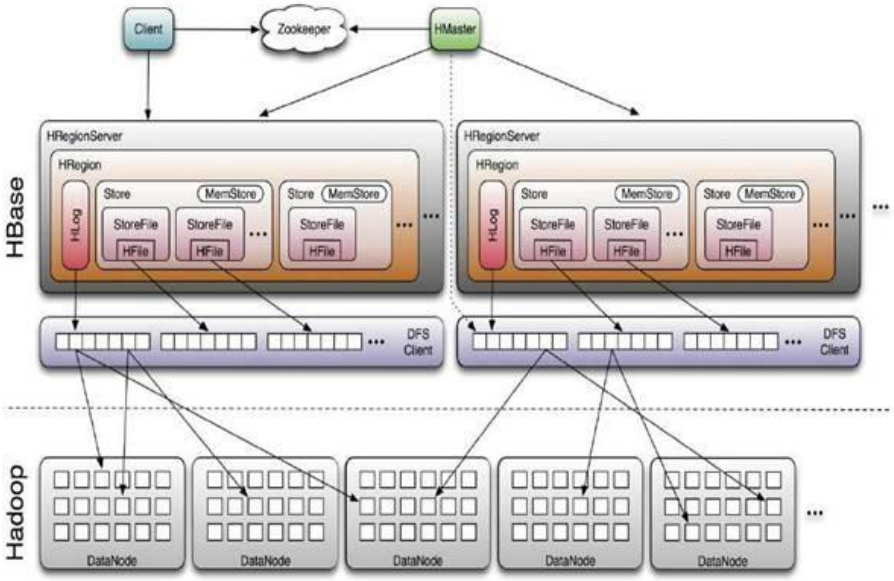

图 7-9 HBase 的系统架构

7.7.1 Client

Client 包含访问 HBase 的接口，client 维护着一些缓存来加快对 HBase 的访问，比如 HRegion 的位置信息。

HBase Client 使用 HBase 的 RPC 机制与 HMaster 和 HRegionServer 进行通信，对于管理类操作，Client 与 HMaster 进行 RPC；对于数据读写类操作，Client 与 HRegionServer 进行 RPC。

7.7.2 Zookeeper

Zookeeper 中除了存储-ROOT-表的地址和 HMaster 的地址，HRegionServer 也会把自己以“Ephemeral”方式注册到 Zookeeper 中，使得 HMaster 可以随时感知到各个 HRegionServer 的健康状态。此外，Zookeeper 也避免了 HMaster 的单点问题。关于 Zookeeper 的作用，这里总结说明如下：

- 保证任何时候，集群中只有一个 HMaster；
- 存储所有 HRegion 的寻址入口；
- 实时监控 HRegionServer 的状态，将 HRegionServer 的上线和下线信息实时通知给 HMaster；
- 存储 HBase 的 schema，包括有哪些表，每个表有哪些列族。

7.7.3 HMaster

HMaster 没有单点问题，HBase 中可以启动多个 HMaster，通过 Zookeeper 的 Master Election 机制保证总有一个 HMaster 运行，HMaster 在功能上主要负责表和 HRegion 的管理工作：

- 管理用户对表的增、删、改、查操作；
- 管理 HRegionServer 的负载均衡，调整 HRegion 分布；
- 在 HRegion 分裂后，负责新 HRegion 的分配；
- 在 HRegionServer 停机后，负责失效 HRegionServer 上的 HRegion 的迁移。

每个 HRegion 服务器都会和 HMaster 服务器通讯，HMaster 的主要任务就是要告诉每个 HRegion 服务器它要维护哪些 HRegion。

HMaster 服务器会和每个 HRegion 服务器保持一个长连接。如果这个连接超时或者断开，会导致：

- HRegion 服务器自动重启。
- HMaster 认为 HRegion 已经死机，同时把它负责的 HRegion 分配到其它 HRegion 服务器。

和 Google 的 BigTable 不同的是，当 BigTable 的 TabletServer 和主服务器通讯中断的情况下，它仍然能提供服务。而 HBase 不能这么做，因为，HBase 没有 BigTable 那样额外的加锁系统，BigTable 是由主服务器管理 TabletServer，同时加锁服务器提供数据访问的，而 HBase 只有唯一一个接入点，就是 HMaster 服务器。

当一个新的 HRegion 服务器登陆到 HMaster 服务器，HMaster 会告诉它先等待分配数据。而当一个 HRegion 死机的时候，HMaster 会把它负责的 HRegion 标记为未分配，然后把它们分配到其它 HRegion 服务器。

7.7.4 HRegionServer

HRegionServer 主要负责响应用户 I/O 请求，向 HDFS 文件系统中读写数据，是 HBase 中最核心的模块（如图 7-10 所示）。

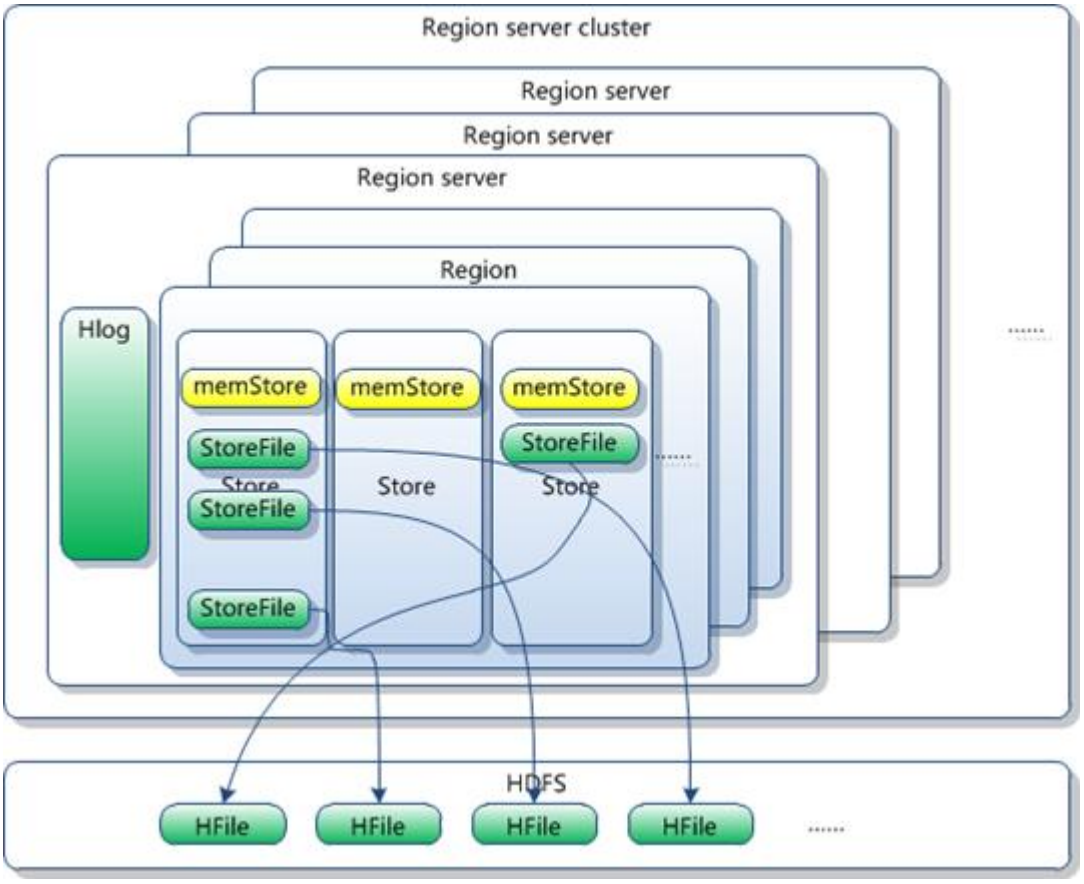

图 7-10 HRegionServer 向 HDFS 文件系统中读写数据

HRegionServer 内部管理了一系列 HRegion 对象，每个 HRegion 对应着表（Table）中的一个 HRegion，HRegion 由多个 HStore 组成。每个 HStore 对应了表中的一个列族的存储，可以看出，每个列族其实就是一个集中的存储单元，因此，最好将具备共同 IO 特性的列放在一个列族中，这样最高效。

Client 访问 HBase 上数据的过程并不需要 HMaster 参与（寻址访问 Zookeeper 和 HRegionServer，数据读写访问 HRegionServer），HMaster 仅仅维护着表和 HRegion 的元数据信息，负载很低。

对于用户来说，每个表是一堆数据的集合，靠主键来区分。物理上，一张表是被拆分成多个块，每一块就称为一个 HRegion。用“表名+开始/结束主键”，来区分一个 HRegion，一个 HRegion 会保存一个表里面某段连续的数据，从开始主键到结束主键，一张完整的表是被分开保存在多个 HRegion 上面的。

所有的 HBase 数据库数据一般是保存在 Hadoop 分布式文件系统 HDFS 上面，用户通过一系列 HRegion 服务器获取这些数据，一般而言，一台机器上面运行一个 HRegion 服务器，而每一个区段 HRegion 只会被一个 HRegion 服务器维护。

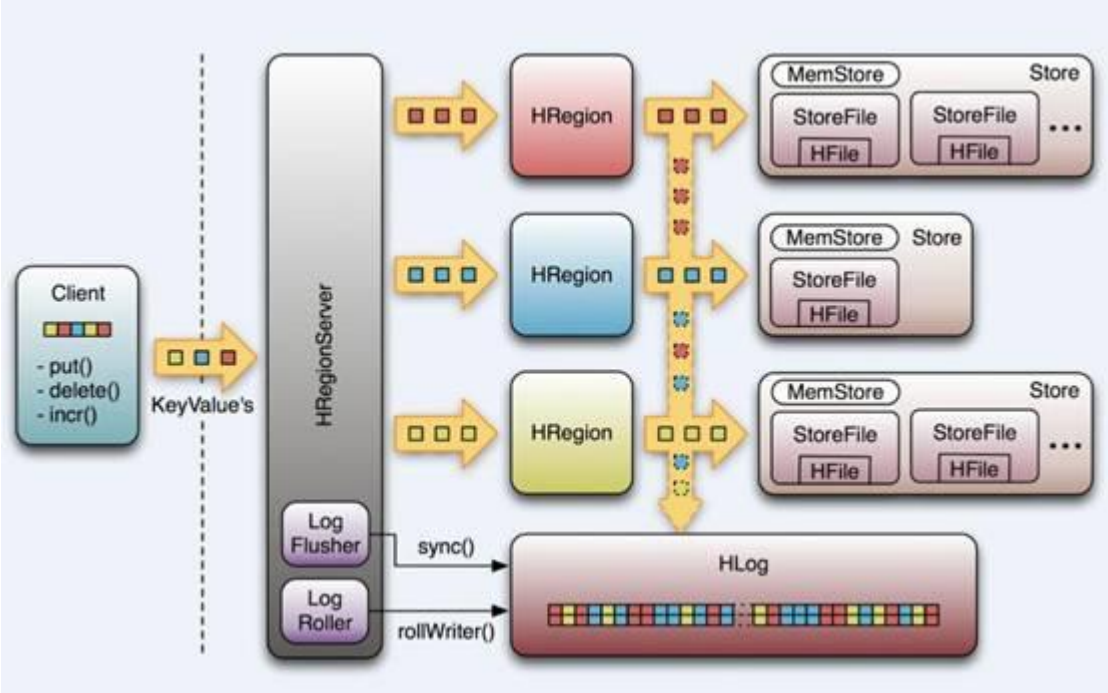

图 7-11 HRegionServer 和 HMemStore 缓存、Hlog 文件

图 7-11 给出了更新和读取数据的过程。当用户需要更新数据的时候，他会被分配到对应的 HRegion 服务器提交修改，这些修改先是被写到 HMemStore 缓存和服务器的 HLog 文件里面，HMemStore 是在内存中的缓存，保存最近更新的数据，HLog 是磁盘上面的记录文件，它记录着所有的更新操作，当操作写入 HLog 之后，commit()调用才会返回给客户端。

当读取数据的时候，HRegion 服务器会先访问 HMemStore 缓存，如果缓存里面没有该数据，才会到 HStore 磁盘上面寻找，每一个列族都会有一个 HStore，每个 HStore 包含很多 HStoreFiles 具体文件，这些文件都是 B 树结构的，方便快速读取。

系统会定期调用 HRegion.flushcache() 把 HMemStore 缓存里面的内容写到文件中，一般这样会增加一个新的 HStoreFile 文件，而此时高速缓存就会被清空，并且写入一个标记到 HLog，表示上面的内容已经被写入到文件中保存。

在启动的时候，每个 HRegion 服务器都会检查自己的 HLog 文件，看看最近一次执行 flushcache 之后有没有新的更新写入操作。如果没有更新，就表示所有数据都已经更新到文件中了；如果有更新，服务器就会先把这些更新写入高速缓存，然后调用 flushcache 写入到文件。最后，服务器会删除旧的 HLog 文件，并开始给用户访问数据。

因此，为了节省时间，可以少去调用 flushcache，但是，这样会增加内存占用，而且，在服务器重启的时候会延长很多时间。如果可以定期调用 flushcache，缓存大小会控制在一个较低的水平，而且，HLog 文件也会很快地重构，但是，调用 flushcache 的时候会造成系

统负载瞬间增加。

HLog 会被定期回滚，回滚的时候是按照时间备份文件，每当回滚的时候，系统会删除那些已经被写到文件中的更新，回滚 HLog 只会占用很少的时间，建议经常回滚以减少文件尺寸。

每一次调用 flushcache 会生成一个新的 HStoreFile 文件，从一个 HStore 里面获取一个值都需要访问所有的 HStoreFile 文件，这样十分耗时，所以，我们要定期把这些分散的文件合并到一个大文件里面，HStore.compact()就可以完成这样的工作。这样的合并工作是十分占用资源的，当 HStoreFile 文件的数量超过一个设定值的时候才会触发。

Google 的 BigTable 有高级合并和低级合并的区别，但是，HBase 没有这个概念，只要记住下面两点就可以了：

（1）flushcache 会建立一个新的 HStoreFile 文件，并把缓存中所有需要更新的数据写到文件里面，flushcache 之后，HLog 的重建次数会清零。

（2）compact 会把所有 HstoreFile 文件合并成一个大文件。

和 BigTable 不同的是，HBase 每个更新如果是被正确提交了并且没有返回错误的话，它就一定是被写到记录文件里面了，这样不会造成数据丢失。

两个 HRegion 可以通过调用 HRegion.closeAndMerge()合并成一个新的 HRegion，当前版本这个操作是需要两台 HRegion 都停机才能操作。

当一个 HRegion 变得太过巨大的时候，超过了设定的阈值，HRegion 服务器会调用 HRegion.closeAndSplit()，这个 HRegion 会被拆分为两个，并且报告给主服务器让它决定由哪个 HRegion 服务器来存放新的 HRegion。这个拆分过程是十分迅速的，因为，两个新的 HRegion 最初只是保留原来 HRegionFile 文件的引用，而这个时候旧的 HRegion 会处于停止服务的状态，当新的 HRegion 构建完成并且把引用删除了以后，旧的 HRegion 才会删除。

最后总结几点：

（1）客户端以表的形式读取数据；

（2）一张表是被划分成多个 HRegion 区域；

（3）HRegion 是被 HRegionServer 管理的，当客户端需要访问某行数据的时候，需要访问对应的 HRegionServer。

（4）HRegionServer 里面有三种方式保存数据：

A、 HMemStore 高速缓存，保留最新写入的数据；

B、 HLog 记录文件，保留的是提交成功了、但未被写入文件的数据；

C、 HStore 文件，数据的物理存放形式。

7.7.5 HStore

HStore 存储是 HBase 存储的核心了，由两部分组成，一部分是 HMemStore，一部分是 HStoreFile。HMemStore 是排序的内存缓冲区，用户写入的数据首先会放入 HMemStore，当 HMemStore 满了以后会 Flush 成一个 HStoreFile（底层实现是 HFile），当 HStoreFile 文件数量增长到一定阈值，会触发合并操作，将多个 HStoreFile 合并成一个 HStoreFile，合并过程中会进行版本合并和数据删除，因此可以看出 HBase 其实只有增加数据，所有的更新和删除操作都是在后续的 compact 过程中进行的，这使得用户的写操作只要进入内存中就可以立即返回，保证了 HBase I/O 的高性能。当 HStoreFile 合并后，会逐步形成越来越大的 HStoreFile，当单个 HStoreFile 大小超过一定阈值后，会触发分裂操作，同时把当前 HRegion 分裂成 2 个 HRegion，父 HRegion 会下线，新分裂出的 2 个孩子 HRegion 会被 HMaster 分配到相应的 HRegionServer 上，使得原先 1 个 HRegion 的压力得以分流到 2 个 HRegion 上。图 7-12 描述了合并和分裂的过程。

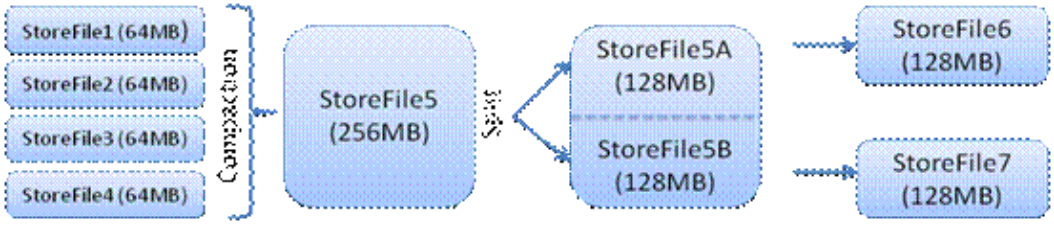

图 7-12 HStoreFile 的合并和分裂过程

在理解了上述 HStore 的基本原理后，还必须了解一下 HLog 的功能，因为，上述的 HStore 在系统正常工作的前提下是没有问题的，但是，在分布式系统环境中，无法避免系统出错或者宕机，因此，一旦 HRegionServer 意外退出，HMemStore 中的内存数据将会丢失，这就需要引入 HLog 了。每个 HRegionServer 中都有一个 HLog 对象，HLog 是一个实现写前日志（Write Ahead Log）的类，在每次用户操作写入 HMemStore 的同时，也会写一份数据到 HLog 文件中（HLog 文件格式见后续），HLog 文件定期会滚动出新的，并删除旧的文件（已持久化到 HStoreFile 中的数据）。当 HRegionServer 意外终止后，HMaster 会通过 Zookeeper 感知到，HMaster 首先会处理遗留的 HLog 文件，将其中不同 HRegion 的 HLog 数据进行拆分，分别放到相应 HRegion 的目录下，然后再将失效的 HRegion 重新分配，领取到这些 HRegion 的 HRegionServer 在加载 HRegion 的过程中，会发现有历史 HLog 需要处理，因此会 Replay

HLog 中的数据到 HMemStore 中，然后刷新到 HStoreFiles，完成数据恢复。

7.7.6 HRegion 分配

任何时刻，一个 HRegion 只能分配给一个 HRegionServer。HMaster 记录了当前有哪些可用的 HRegionServer，以及当前哪些 HRegion 分配给了哪些 HRegionServer，哪些 HRegion 还没有分配。当存在未分配的 HRegion 时，并且有一个 HRegionServer 上有可用空间时，HMaster 就给这个 HRegionServer 发送一个装载请求，把 HRegion 分配给这个 HRegionServer。HRegionServer 得到请求后，就开始对此 HRegion 提供服务。

7.7.7 HRegionServer 上线

HMaster 使用 Zookeeper 来跟踪 HRegionServer 的状态。当某个 HRegionServer 启动时，会首先在 Zookeeper 上的 server 目录下建立代表自己的文件，并获得该文件的独占锁。由于 HMaster 订阅了 server 目录上的变更消息，当 server 目录下的文件出现新增或删除操作时，HMaster 可以得到来自 Zookeeper 的实时通知。因此，一旦 HRegionServer 上线，HMaster 能马上得到消息。

7.7.8 HRegionServer 下线

当 HRegionServer 下线时，它和 Zookeeper 的会话断开，Zookeeper 而自动释放代表这台 server 的文件上的独占锁。而 HMaster 不断轮询 server 目录下文件的锁状态。如果 HMaster 发现某个 HRegionServer 丢失了它自己的独占锁(或者 HMaster 连续几次和 HRegionServer 通信都无法成功)，HMaster 就尝试去获取代表这个 HRegionServer 的读写锁，一旦获取成功，就可以确定下面两种情形中的一种发生了：

- HRegionServer 和 Zookeeper 之间的网络断开了；
- HRegionServer 挂了。

无论哪种情况，HRegionServer 都无法继续为它的 HRegion 提供服务了，此时 HMaster 会删除 server 目录下代表这台 HRegionServer 的文件，并将这台 HRegionServer 的 HRegion 分配给其它还活着的 HRegionServer。

如果网络短暂出现问题导致 HRegionServer 丢失了它的锁，那么 HRegionServer 重新连

接到 Zookeeper 之后，只要代表它的文件还在，它就会不断尝试获取这个文件上的锁，一旦获取到了，就可以继续提供服务。

7.7.9 HMaster 上线

HMaster 启动时，需要执行以下步骤：

- 第一步：从 Zookeeper 上获取唯一一个代表该 HMaster 的锁，用来阻止其它 HMaster 成为主服务器；
- 第二步：扫描 Zookeeper 上的 server 目录，获得当前可用的 HRegionServer 列表；
- 第三步：和第二步中的每个 HRegionServer 通信，获得当前已分配的 HRegion 和 HRegionServer 的对应关系；
- 第四步：扫描.META.中 HRegion 的集合，计算得到当前还未分配的 HRegion，将他们放入待分配 HRegion 列表。

7.7.10 HMaster 下线

由于 HMaster 只维护表和 HRegion 的元数据，而不参与表数据 IO 的过程，HMaster 下线，仅导致所有元数据的修改被冻结(无法创建删除表，无法修改表的 schema，无法进行 HRegion 的负载均衡，无法处理 HRegion 上下线，无法进行 HRegion 的合并，唯一例外的是 HRegion 的分裂可以正常进行，因为只有 HRegionServer 参与)，表的数据读写还可以正常进行。因此，HMaster 下线短时间内对整个 HBase 集群没有影响。从上线过程可以看到，HMaster 保存的信息全是可以冗余信息（都可以从系统其它地方收集到或者计算出来），因此，一般 HBase 集群中总是有一个 HMaster 在提供服务，还有一个以上的 HMaster 在等待时机抢占它的位置。

7.8 HBase 存储格式

HBase 中的所有数据文件都存储在 Hadoop 分布式文件系统 HDFS 上，主要包括上述提出的两种文件类型：

- **HFile**：HBase 中 KeyValue 数据的存储格式，HFile 是 Hadoop 的二进制格式文件，实际上 HStoreFile 就是对 HFile 做了轻量级包装，即 HStoreFile 底层就是 HFile。

- **HLog File**：HBase 中 WAL（Write Ahead Log）的存储格式，物理上是 Hadoop 的顺序文件。

7.8.1 HFile

图 7-13 描述了 HFile 的存储格式。

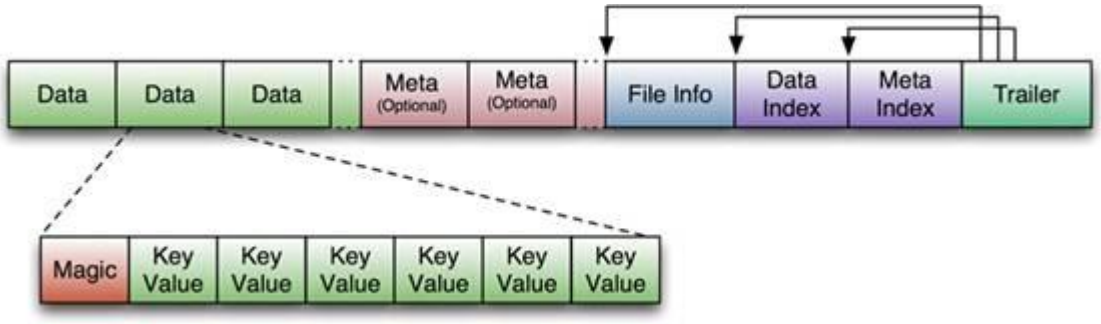

图 7-13 HFile 的存储格式

HFile 分为六个部分：

- Data Block 段：保存表中的数据，这部分可以被压缩；
- Meta Block 段 (可选的)：保存用户自定义的 key/value 对，可以被压缩；
- File Info 段：HFile 的元信息，不被压缩，用户也可以在这一部分添加自己的元信息；
- Data Block Index 段：Data Block 的索引。每条索引的 key 是被索引的 block 的第一条记录的 key；
- Meta Block Index 段(可选的)：Meta Block 的索引；
- Trailer：这一段是定长的。保存了每一段的偏移量，读取一个 HFile 时，会首先读取 Trailer，Trailer 保存了每个段的起始位置(段的 Magic Number 用来做安全 check)，然后，DataBlock Index 会被读取到内存中，这样，当检索某个 key 时，不需要扫描整个 HFile，而只需从内存中找到 key 所在的 block，通过一次磁盘 IO 将整个 block 读取到内存中，再找到需要的 key。DataBlock Index 采用 LRU 机制淘汰。

HFile 的 Data Block 和 Meta Block 通常采用压缩方式存储，压缩之后可以大大减少网络 IO 和磁盘 IO，随之而来的开销当然是需要花费 CPU 进行压缩和解压缩。目标 HFile 的压缩支持两种方式：Gzip 和 LZO。

HFile 文件是不定长的，长度固定的只有其中的两块：Trailer 和 FileInfo。正如图 7-13 中所示的，Trailer 中有指针指向其他数据块的起始点。File Info 中记录了文件的一些 Meta 信息，例如：AVG_KEY_LEN， AVG_VALUE_LEN， LAST_KEY， COMPARATOR，MAX_SEQ_ID_KEY 等。Data Index 和 Meta Index 块记录了每个 Data 块和 Meta 块的起始点。

Data Block 是 HBase I/O 的基本单元，为了提高效率，HRegionServer 中有基于 LRU 的 Block Cache 机制。每个 Data 块的大小可以在创建一个表的时候通过参数指定，大号的 Block 有利于顺序扫描，小号 Block 利于随机查询。每个 Data 块除了开头的 Magic 以外就是一个个 KeyValue 对拼接而成， Magic 内容就是一些随机数字，目的是防止数据损坏。后面会详细介绍每个 Key/Value 对的内部构造。

HFile 里面的每个 Key/Value 对就是一个简单的 byte 数组。但是这个 byte 数组里面包含了很多项，并且有固定的结构（如图 7-14 所示）。

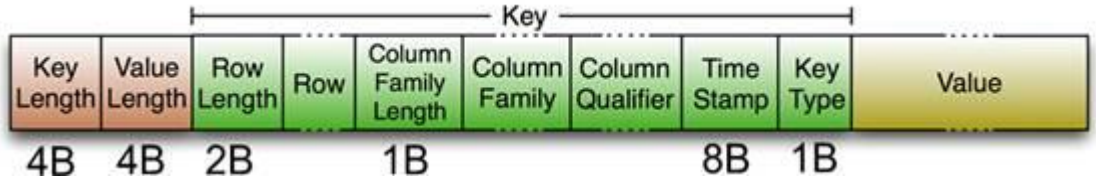

图 7-14 Hfile 里面的 Key/Value 的具体结构

开始是两个固定长度的数值，分别表示 Key 的长度和 Value 的长度。紧接着是 Key，开始是固定长度的数值，表示 RowKey 的长度，紧接着是 RowKey，然后是固定长度的数值，表示 Family 的长度，然后是 Family，接着是 Qualifier，然后是两个固定长度的数值，表示 Time Stamp 和 Key Type（Put/Delete）。Value 部分没有这么复杂的结构，就是纯粹的二进制数据了。

7.8.2 HLogFile

HLog 又称 WAL。WAL 意为 Write Ahead Log，类似 Mysql 中的 binlog，用来做灾难恢复，HLog 记录数据的所有变更，一旦数据修改，就可以从 HLog 中进行恢复。

每个 HRegion Server 维护一个 HLog，而不是每个 HRegion 一个。这样不同 HRegion(来自不同表)的日志会混在一起，这样做的目的是，不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此，可以提高对表的写性能。带来的麻烦是，如果一台

HRegionServer 下线，为了恢复其上的 HRegion，需要将 HRegionServer 上的 HLog 进行拆分，然后分发到其它 HRegionServer 上进行恢复。

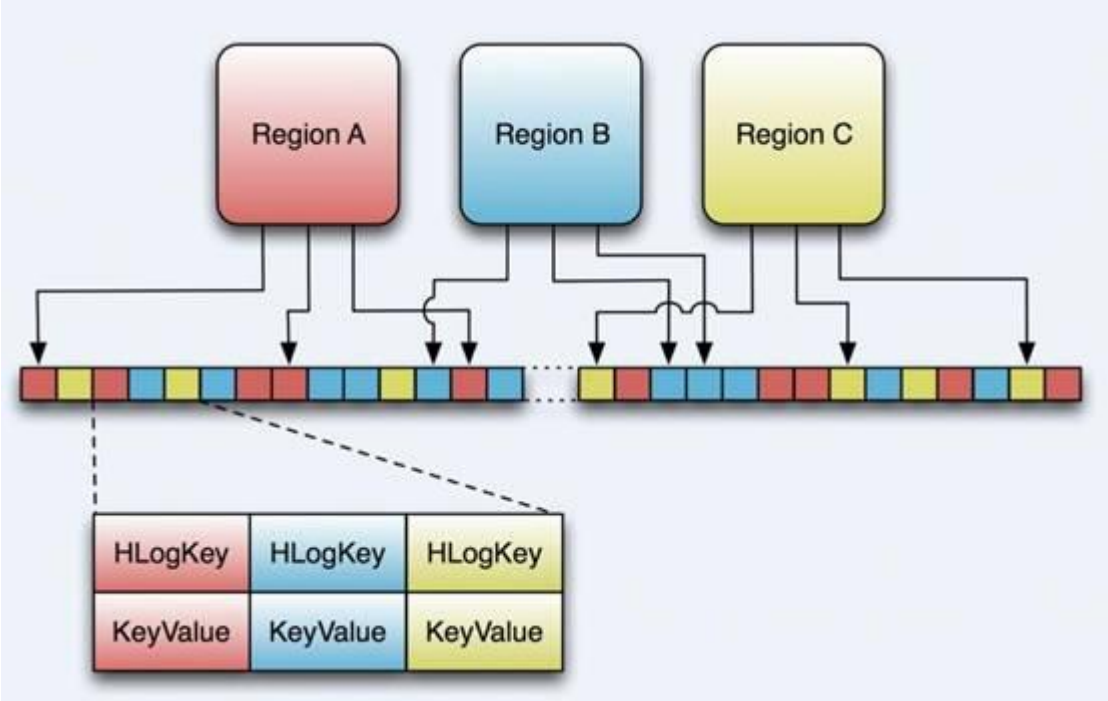

图 7-15 Hlog 文件的结构

图 7-15 给出了 HLog 文件的结构，其实，HLog 文件就是一个普通的 Hadoop 顺序文件（Sequence File），顺序文件的 Key 是 HLogKey 对象，HLogKey 中记录了写入数据的归属信息，除了表和 HRegion 名字外，同时还包括顺序号和时间戳，时间戳是“写入时间”，顺序号的起始值为 0，或者是最近一次存入文件系统中顺序号。HLog 顺序文件的 Value 是 HBase 的 Key/Value 对象，即对应 HFile 中的 Key/Value。

7.9读写数据

HBase 使用 HMemStore 和 HStoreFile 存储对表的更新。

数据在更新时，首先写入 HLog 和内存(HMemStore)中，HMemStore 中的数据是排序的，当 HMemStore 累计到一定阈值时，就会创建一个新的 HMemStore，并且将老的 HMemStore 添加到 flush 队列，由单独的线程 flush 到磁盘上，成为一个 HStoreFile。与此同时，系统会在 Zookeeper 中记录一个检查点，表示这个时刻之前的变更已经持久化了。

当系统出现意外时，可能导致内存(HMemStore)中的数据丢失，此时使用 HLog 来恢复

检查点之后的数据。

HStoreFile 是只读的，一旦创建后就不可以再修改。因此，HBase 的更新其实是不断追加的操作。当一个 HStore 中的 HStoreFile 达到一定的阈值后，就会进行一次合并，将对同一个 key 的修改合并到一起，形成一个大的 HStoreFile，当 HStoreFile 的大小达到一定阈值后，又会对 HStoreFile 进行分裂，等分为两个 HStoreFile。

由于对表的更新是不断追加的，处理读请求时，需要访问 HStore 中全部的 HStoreFile 和 HMemStore，将它们按照行键进行合并，由于 HStoreFile 和 HMemStore 都是经过排序的，并且 HStoreFile 带有内存中索引，合并的过程还是比较快的。

写请求处理过程具体如下：

- client 向 HRegionServer 提交写请求；
- HRegionServer 找到目标 HRegion；
- HRegion 检查数据是否与 schema 一致；
- 如果客户端没有指定版本，则获取当前系统时间作为数据版本；
- 将更新写入 HLog；
- 将更新写入 HMemstore；
- 判断 HMemStore 是否需要 flush 为 HStore 文件。

7.10 MapReduce on HBase

在 HBase 系统上运行批处理运算，最方便和实用的模型依然是 MapReduce，如图 7-16 所示。

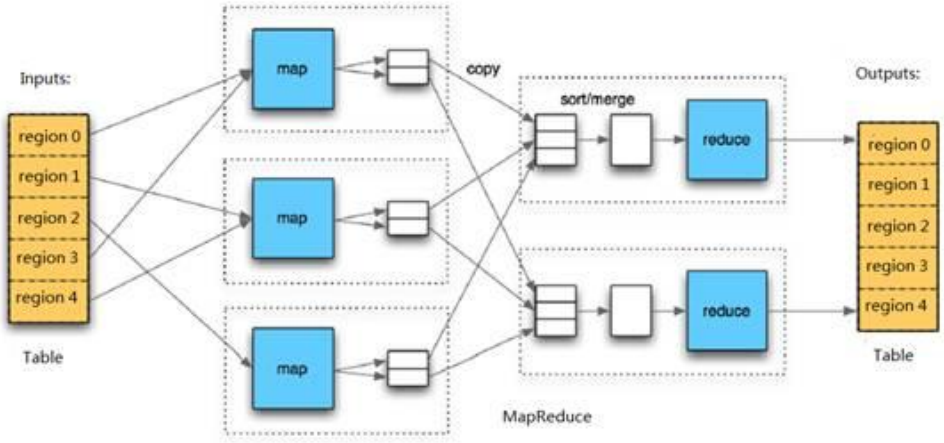

图 7-16 MapReduce 过程示意图

HBase Table 和 HRegion 的关系，比较类似 HDFS File 和 Block 的关系，HBase 提供了配套的 TableInputFormat 和 TableOutputFormat API，可以方便地将 HBase Table 作为 Hadoop MapReduce 的 Source 和 Sink，对于 MapReduce Job 应用开发人员来说，基本不需要关注 HBase 系统自身的细节。

本章小结

本章介绍了 HBase 的相关知识、使用场景和成功案例，并介绍了 HBase 和传统关系数据库的对比分析；接下来介绍了 HBase 访问接口、数据模型、实现方法、系统架构、存储格式、读写数据等；最后，简单介绍了 MapReduce 是在 HBase 系统上运行批处理运算的最方便和实用的模型。

参考文献

[1] HBase. 百度百科.

[2] HBase 分析报告. 百度文库. http://wenku.baidu.com/view/cf4b96c4bb4cf7ec4afed07d.html

第 8 章 流计算

随着大数据时代的到来，数据量急剧膨胀，业务也变得越加复杂。在业务中产生了源源不断的数据流，而数据的价值又随着时间的流逝而降低，如何实时处理海量流数据成为一大挑战。传统的数据库方案已不适合处理这样的数据，而流计算则可以持续地对流数据进行分析，实时得出有价值的信息。

本章内容首先介绍了什么是流计算，包括概念、处理模型和处理流程，并详细介绍了当前热门的开源流计算框架 Storm，内容要点如下：

- 流计算概述
- 流计算处理流程
- 流计算应用
- 流计算框架 Storm

8.1 流计算概述

8.1.1 什么是流计算

近年来，一种新的数据密集型应用已经得到了广泛的认同，这类应用的特征是：数据不宜用持久稳定的关系型模型建模，而适宜用瞬态数据流建模。这些应用的实例包括金融服务、网络监控、电信数据管理、Web 应用、生产制造、传感检测等等。在这种数据流模型中，单独的数据单元可能是相关的元组，例如网络测量、呼叫记录、网页访问等产生的数据。但是，这些数据以大量、快速、时变（可能是不可预知的）的数据流形式持续到达，由此产生了一些基础性的新的研究问题。

互联网从诞生的第一时间起，对世界的最大的改变就是让信息能够实时交互，数据库和高速网络的发展更是给互联网业务带来了实时性的改变。对于实时性要求很高的应用，若把持续到达的数据简单地放到 DBMS 中，再在其中进行操作，是不太现实的。传统的 DBMS 并不是为快速连续的存放单独的数据单元而设计的，而且也并不支持“持续处理”，而“持续处理”是数据流应用的典型特征。

随着大数据时代的到来，互联网业务的发展从初期数据量小、业务简单，到过渡期数据有所膨胀、业务较复杂，再到如今大数据时期数据量急剧膨胀，业务很复杂的情况。面对大

数据，特别是流数据的实时化需求，传统的数据库技术方案已不能满足需求，成本高的同时并不能带来高效率，急需针对流数据的实时计算——流计算。

流计算，即针对流数据的实时计算。要了解流计算，就要了解两个概念：流数据和实时计算。

流数据，也称流式数据，是指将数据看作数据流的形式来处理。数据流是在时间分布和数量上无限的一系列动态数据集合体；数据记录是数据流的最小组成单元。数据流具有如下特征：

- 数据连续不断；
- 数据来源众多，格式复杂；
- 数据量大，但是不十分关注存储；
- 注重数据的整体价值，不要过分关注个别数据；
- 数据流顺序颠倒，或者不完整。

实时计算是针对大数据而言的。对于少量数据而言，实时计算并不存在问题，但随着数据量的不断膨胀，实时计算就发生了质的改变，数据的结构与来源越来越多样化，实时计算的逻辑也变得越来越复杂。除了像非实时计算的需求（如计算结果准确）以外，实时计算最重要的一个需求是能够实时响应计算结果，一般要求为秒级。

图 8-1 是一个流计算的示意图：实时获取来自不同终端的海量数据，经过流计算平台的不断地分析处理，整合获得有价值的信息。

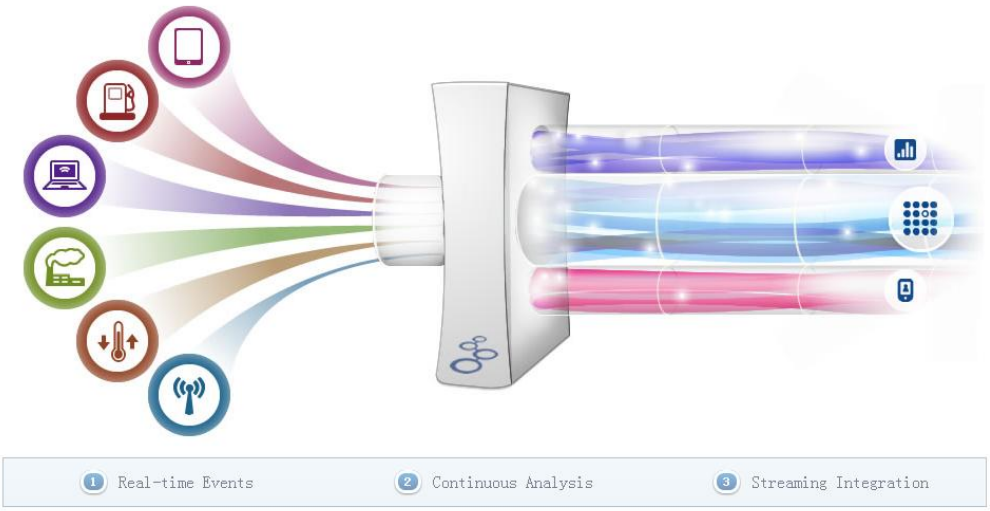

图 8-1 流计算示意图

总的来说，流计算来自一个信念：数据的价值随着时间的流逝而降低。所以，事件出现后必须尽快地对它们进行处理，最好数据出现时便立刻对其进行处理。发生一个事件就进行

一次处理，而不是缓存起来成一批再处理。例如商用搜索引擎，像 Google、Bing 和 Yahoo！等，通常在用户查询响应中提供结构化的 Web 结果，同时也插入基于流量的点击付费模式的文本广告。为了在页面上最佳位置展现最相关的广告，就需要对用户数据进行实时分析，通过一些算法来动态估算给定上下文中一个广告被点击的可能性，从而能展示更佳的广告。为了及时处理用户反馈，需要一个低延迟、可扩展、高可靠的处理引擎。

8.1.2 数据流与传统的关系存储模型的区别

在数据流模型中，需要处理的输入数据（全部或部分）并不存储在可随机访问的磁盘或内存中，而是以一个或多个“连续数据流”的形式到达。数据流不同于传统的关系存储模型，主要区别有如下几个方面：

- 流中的数据元素在线到达；
- 系统无法控制将要处理的新到达的数据元素的顺序；
- 数据流的潜在大小也许是无穷无尽的；
- 一旦数据流中的某个元素经过处理，要么被丢弃，要么被归档存储。因此，除非该数据被直接存储在内存中，否则将不容易被检索。相对于数据流的大小，这是一种典型的极小相关。

8.1.3 流计算需求

对于一个流计算系统来说，它应达到如下需求：

- 高性能：处理大数据的基本要求，如每秒处理几十万条数据。
- 海量式：支持 TB 级甚至是 PB 级的数据规模。
- 实时性：必须保证一个较低的延迟时间，达到秒级别，甚至是毫秒级别。
- 分布式：支持大数据的基本架构，必须能够平滑扩展。
- 易用性：能够快速进行开发和部署。
- 可靠性：能可靠地处理流数据。

针对不同的应用场景，相应的流计算系统会有不同的需求，但是，针对海量数据的流计算，无论在数据采集、数据处理中都应达到秒级别的要求。

8.1.4 流计算与 Hadoop

谈到大规模数据的处理，很容易想到 Hadoop 和 MapReduce。Hadoop 是大数据分析领域的王者，那么 MapReduce 模式能否胜任实时流计算系统的需求呢？

Hadoop 在本质上是一个批处理系统。数据被引入 Hadoop 文件系统 (HDFS) 并分发到各个节点进行处理。当处理完成时，结果数据返回到 HDFS 供始发者使用。Hadoop 的批量化处理是人们喜爱它的地方，但这在某些领域仍显不足，尤其是在例如移动、Web 客户端或金融、网页广告等需要实时计算的领域。这些领域产生的数据量极大，没有足够的存储空间来存储每个业务收到的数据。而流计算则可以实时对数据进行分析，并决定是否抛弃无用的数据，而这无需经过 Map/Reduce 的环节。

为了保证实时性，许多实时数据流处理系统都是专用系统，它们不得不面对可靠性、扩展性和伸缩性方面的问题。使用 MapReduce 的好处在于 Hadoop 帮助业务屏蔽了底层处理，上层作业不用关心容错和扩容方面的问题，应用升级也很方便。不过基于 MapReduce 的业务不得不面对处理延迟的问题。有一种想法是将基于 MapReduce 的批量处理转为小批量处理，将输入数据切成小的片段，每隔一个周期就启动一次 MapReduce 作业，这种实现需要减少每个片段的延迟，并且需要考虑系统的复杂度：

- 将输入数据分隔成固定大小的片段，再由 MapReduce 平台处理，缺点在于处理延迟与数据片段的长度、初始化处理任务的开销成正比。小的分段是会降低延迟，但是，也增加附加开销，并且分段之间的依赖管理更加复杂（例如一个分段可能会需要前一个分段的信息）；反之，大的分段会增加延迟。最优化的分段大小取决于具体应用。
- 为了支持流式处理，MapReduce 需要被改造成 Pipeline 的模式，而不是 reduce 直接输出；考虑到效率，中间结果最好只保存在内存中等等。这些改动使得原有的 MapReduce 框架的复杂度大大增加，不利于系统的维护和扩展。
- 用户被迫使用 MapReduce 的接口来定义流式作业，这使得用户程序的可伸缩性降低。

MapReduce 框架为批处理做了高度优化，系统典型地通过调度批量任务来操作静态数据，任务不是常驻服务，数据也不是实时流入；而数据流计算的典型范式之一是不确定数据速率的事件流流入系统，系统处理能力必须与事件流量匹配。数据流实时处理的模式决定了要和批处理使用非常不同的架构，试图搭建一个既适合流式计算又适合批处理的通用平台，

结果可能会是一个高度复杂的系统，并且最终系统可能对两种计算都不理想。

如 Facebook 通过对 Hadoop/HBase 进行实时化改造，使其具有了一定的实时处理能力（可参阅 Facebook 发布的论文《Apache Hadoop Goes Realtime at Facebook》），但这并不能算是一个较好的通用流计算解决方案。

因此，当前业界诞生了许多专门的数据流实时计算系统来满足各自需求，当然除了延迟，它们需要解决可靠性、扩展性和伸缩性等方面的挑战。

8.2 流计算处理流程

传统的数据操作（如图 8-2 所示），首先将数据采集并存储在 DBMS 中，然后通过查询和 DBMS 进行交互，得到用户想要的结果。这样的一个流程隐含了两个前提：

- 数据是旧的。当对数据做查询的时候，里面数据已经是过去某一个时刻数据的一个快照，这些数据可能已经过期了；
- 这样的流程需要人们主动发出查询。也就是说**用户是主动的，而 DBMS 系统是被动的**。

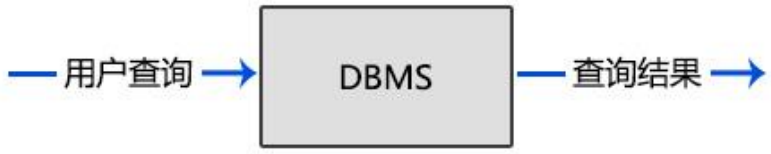

图 8-2 传统数据操作流程

对于流计算（如图 8-3 所示），其数据的处理流程一般有三个阶段：数据实时采集、数据实时计算、实时查询服务。

图 8-3 流计算的数据处理流程

8.2.1 数据实时采集

在数据实时采集阶段，由于现在分布式集群得到广泛应用，数据可能分散存储在不同的机器上，要处理这些数据，首先就要进行一个实时采集的过程，汇总来自不同机器上的数据。数据的实时采集要保证实时性、低延迟与稳定可靠。

目前有许多优秀的开源分布式日志收集系统均可满足每秒数百 MB 的数据采集和传输需求。如 Hadoop 的 Chukwa、Facebook 的 Scribe、LinkedIn 的 Kafka、Cloudera 的 Flume、淘宝的 TimeTunnel 等。

一般来说，数据采集系统基本架构有三个部分（如图 8-4 所示）：

- Agent: 主动采集数据，并把数据推送到 collector；
- Collector: 接收多个 Agent 的数据，并实现有序、可靠、高性能的转发；
- Store: 存储 Collector 的数据。

但对于流计算，一般在 Store 部分不进行存储，而是直接发送给流计算平台进行计算。

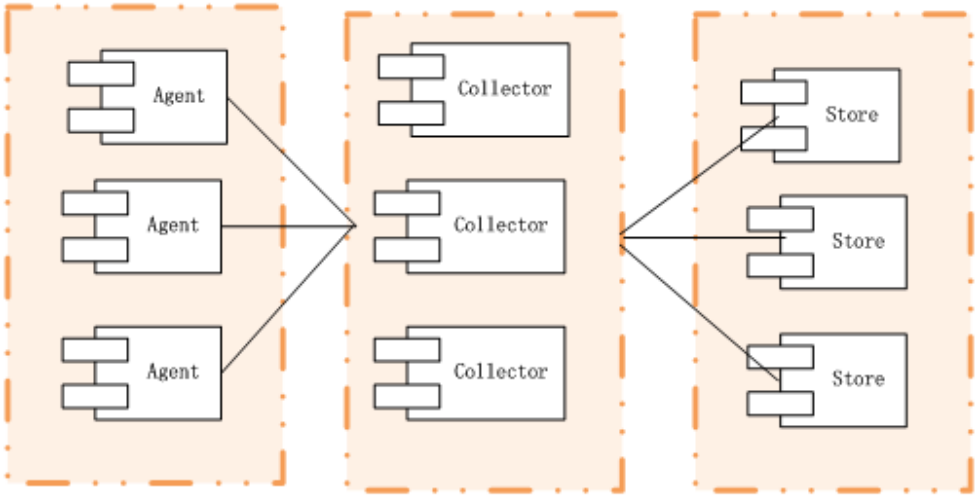

图 8-4 数据实时采集系统基本架构

8.2.2 数据实时计算

如图 8-5 所示，接收数据采集系统源源不断发来的实时数据后，流计算系统在流数据不断变化的运动过程中实时地进行分析，捕捉到可能对用户有用的信息，并把结果发送出去。数据实时计算与传统的数据操作的不同之处包括以下两个方面：

- 能对流数据做出实时回应；
- 用户是被动的，而 DBMS 是主动的。

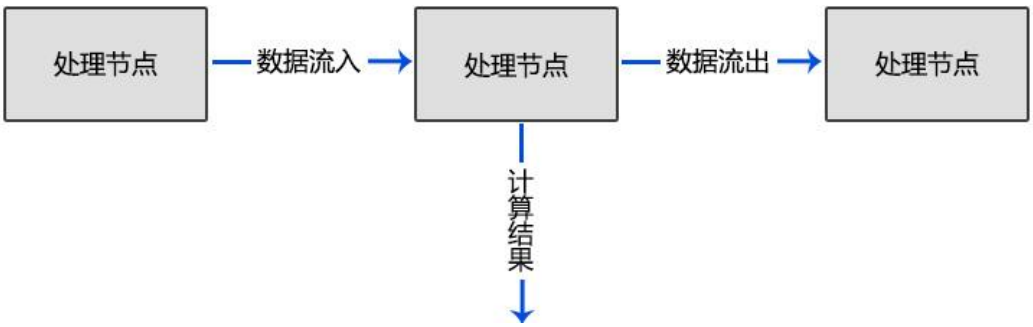

图 8-5 数据实时计算流程

实时数据经过处理节点后，产生的结果可作为另一个处理节点的输入数据，直至获取所需的计算结果。处理节点也可以将这些数据保存下来，以便下一阶段使用。

8.2.3 实时查询服务

理想情况下，流计算会将对用户有价值的结果实时推送给用户，这取决于应用场景。一

般而言，流计算的第三个阶段是实时查询服务，经由流计算框架得出的结果可供用户进行实时查询、展示或储存。

8.3 流计算的应用

流计算是针对流数据的实时计算，主要应用在产生大量流数据并对实时性要求很高的领域。

8.3.1 流计算的应用场景

如对于大型网站，活跃的流式数据非常普遍，这些数据包括网站的访问 PV（page view）/UV（unique visitor）、用户访问了什么内容、搜索了什么内容等。实时的数据计算和分析可以动态展示网站实时流量的变化情况，分析每天各小时的流量和用户分布情况，这对于大型网站来说具有重要的实际意义，不仅可用于网站的实时业务监控，也可以实现用户实时个性化内容推荐等。

流计算的应用场景有很多，总的来说，流计算一方面可应用于处理金融服务如股票交易、银行交易等产生的大量实时数据。另一方面流计算主要应用于各种实时 Web 服务中，如搜索引擎、购物网站的实时广告推荐，SNS 社交类网站的实时个性化内容推荐，大型网站、网店的实时用户访问情况分析等。

但从另一方面来说，并不是每个应用场景都需要实时流计算的。需要考虑是否对数据的实时性有迫切需求、是否更关注对当前数据的分析与响应。若处理动态流程（其特征更改得相当频繁）、非线性流程（其计时和顺序不可预测）和需要实时响应外部事件的流程，则流计算是最适合的。

8.3.2 流计算实例

（1）量子恒道

流计算的一大应用领域是分析系统。传统的分析系统都是分布式离线计算的方式，即将数据全部保存起来，然后每隔一定的时间进行离线分析，从而得出结果。但这样必然会导致一定的延时，这取决于离线计算的间隔时间和计算时长。特别是对于海量数据而言，即使是短时间内就可计算出结果，但离线计算间隔时间过长的话延时也相应增加。

但是，随着业务对实时性要求的提升，这样的模式已不太适合对于流数据的分析，也不太适用于需要实时响应的互联网应用场景。而通过流计算，能在秒级别内得到实时的分析结果，有利于根据当前所得到的分析结果及时地做出决策、调整。典型的搜索引擎、购物网站的广告推荐、社交网站的个性化推荐等，都是基于对用户行为的分析系统实现的。典型的代表还有网站访问数据的分析。接下来我们将以量子恒道的例子来说明流计算给分析系统带来的改变。

量子恒道是一家专业电子商务数据服务商，致力于为网商提供精准实时的数据统计、多维的数据分析、权威的数据解决方案。目前为超过百万的淘宝卖家提供数据统计分析服务。

随着用户和访问数据规模的不断增加，量子恒道也面临着巨大的挑战：实时计算处理数据超过 3T/日，分布式离线计算处理数据超过 20T/日。虽然分布式离线计算能满足大部分用户的需求，小时级的统计延时是可以接受的。但随着实时性要求的不断提升，特别是“双 11”、“双 12”这样需要实时数据分析支撑的应用场景，商家希望通过实时的网店访问情况来及时调整促销策略。如何实现秒级别的实时分析响应成为量子恒道的一大挑战。

网站访问数据是典型的流数据，针对流数据，量子恒道基于“Erlang（一种通用的面向并发的编程语言）+ZooKeeper（针对大型分布式系统的可靠协调系统）”开发了海量数据实时流计算框架 Super Mario 2.0。该流计算框架具有低延迟、高可靠性的特点。

与前面介绍的流计算的三个阶段相对应，Super Mario 2.0 的实时数据处理流程也可以用以下三个阶段来表示（如图 8-6 所示）：

- Log 数据由 TimeTunnel 在毫秒级别内实时送达；
- 实时数据经由 Super Mario 流计算框架进行处理；
- HBase 输出、存储结果。

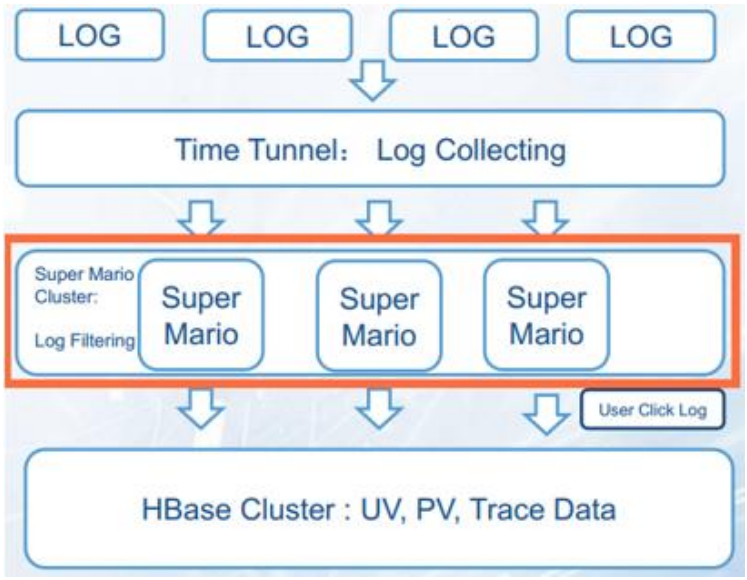

图 8-6 Super Mario 实时数据处理流程

通过 Super Mario 流计算框架，量子恒道可处理每天 TB 级的实时流数据，并且从用户发出请求到数据展示，整个延时控制在 2-3 秒内，达到了实时性的要求。

（2）IBM InfoSphere Streams

流计算不仅为互联网带来改变，也能改变我们的生活。IBM 的流计算平台 InfoSphere Streams（如图 8-7 所示），能够广泛应用于制造、零售、交通运输、金融证券以及监管各行各业的解决方案之中，使得实时快速做出决策的理念得以实现。以实时交通信息管理为例，Streams 应用于斯德哥尔摩的交通信息管理，通过结合来自不同源的实时数据，Streams 可以生成动态的、多方位的看待交通流量的方式，为城市规划者和乘客提供实时交通状况查看。

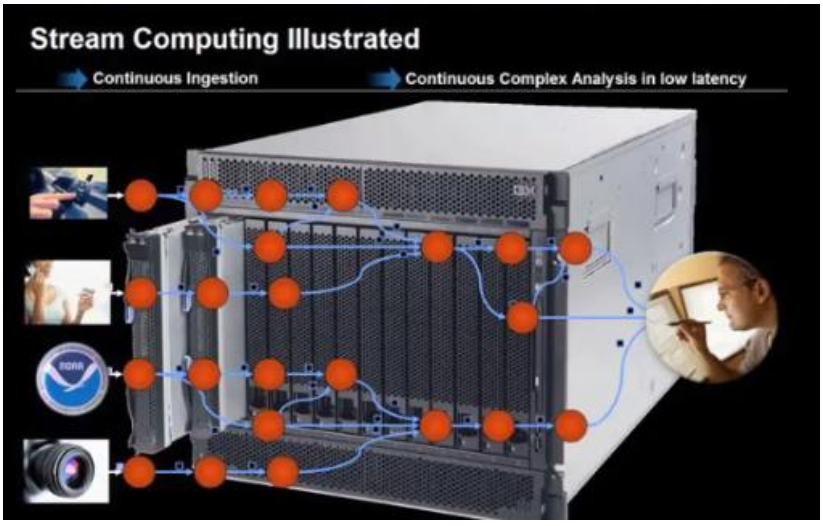

图 8-7 Streams 可汇总来自不同源的实时数据

通过实时流计算来分析交通信息是很有现实意义的。以提供导航路线为例，一般的导航路线计算并没有考虑交通状况，因为要处理如此庞大的实时信息就是一个极大的挑战。即便

计算路线时有考虑交通状况，往往也只是参考了以往的交通状况。而借助于实时流计算，不仅可以根据交通情况制定路线，而且在行驶过程中，也可以根据交通情况的变化实时更新路线，始终为用户提供最佳的行驶路线，如图 8-8 所示。

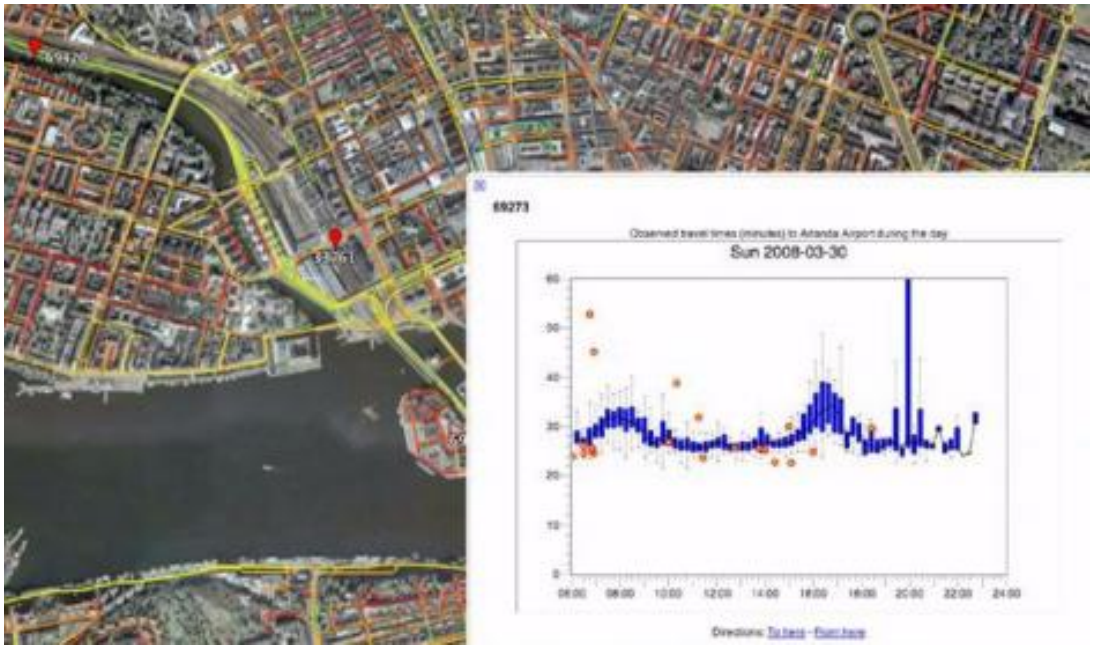

图 8-8 通过 Streams 分析实时交通信息

8.4 流计算框架 Storm

自从数据流出现以来，就有分析数据流并从中获取有用信息的需求。但是，直到几年前，仍然只有那些非常大的银行和政府机构能够通过昂贵的定制系统满足这种计算需求，如 IBM 推出的商业流计算系统 InfoSphere Streams，在政府部分与金融机构得以使用。

早在 InfoSphere Streams 出现之前，就有许多流计算技术的学术研究，如 Aurora，它是 MIT 等三所大学合作完成的项目。后来在 Aurora 的基础上开发了流式系统 Borealis，但该项目在 08 年已经停止维护。

流数据一般出现在金融行业或者互联网流量监控的业务场景，由于这些场景中数据库应用占据主导地位，因而造成了早期对于流数据研究多是基于对传统数据库处理的流式化，而对流式框架本身的研究则偏少，当时的工业界把更多的精力转向了实时数据库。

2010 年 Yahoo!开发的分布式流式处理系统 S4（Simple Scalable Streaming System）的开源，以及 2011 年 Twitter 开发的 Storm 的开源，改变了这个情况。S4 和 Storm 相比 Hadoop 而言，在流数据处理上更具优势。MapReduce 系统主要解决的是对静态数据的批量处理，

即当前的 MapReduce 系统实现启动计算时，一般数据已经到位。而流式计算系统在启动时，一般数据并没有完全到位，而是源源不断地流入。批处理系统一般重视数据处理的总吞吐量，而流处理系统则更加关注数据处理的延时，即希望流入的数据越快处理越好。

以往开发人员在做一个实时应用的时候，除了要关注应用逻辑计算处理本身，还要为了数据的实时传输、交互、分布大伤脑筋，但是，现在情况却大为不同。以 Storm 为例，开发人员可以快速地搭建一套健壮、易用的实时流处理框架，配合 SQL 产品或者 NoSQL 产品或者 MapReduce 计算平台，就可以低成本地做出很多以前很难想象的实时产品。

Yahoo! S4 与 Twitter Storm 是目前流行的开源流计算框架，各有其架构特点，相对而言，Storm 更为优秀。我们在此就以 Storm 为研究学习对象，学习其设计理念与架构特点。

8.4.1 Storm 简介

Twitter Storm 是一个免费、开源的分布式实时计算系统，它可以简单、高效、可靠地处理大量的流数据。Storm 对于实时计算的意义类似于 Hadoop 对于批处理的意义，这一说法也得到了业内人士的认同。

Storm 是基于 Clojure 和 Java 开发的，可以访问其官方网站 http://storm-project.net/或 Github 项目主页 https://github.com/nathanmarz/storm 了解其更多信息。

Twitter 开发这样一款系统也是为了应对其不断增长的数据和实时处理需求。为了处理最近的数据，需要一个实时系统和批处理系统同时运行。当要计算一个查询时，需要查询批处理视图和实时视图，并把它们合并起来以得到最终的结果。

在 Twitter 中进行实时计算的系统就是 Storm，它在数据流上进行持续计算，并且对这种流式数据处理提供了有力保障。

同时，Twitter 采用分层的数据处理架构（如图 8-9 所示），由 Hadoop 和 ElephantDB（专门用于从 Hadoop 中导出 key/value 数据的数据库）组成批处理系统，Storm 和 Cassandra（混合型的非关系的数据库）组成实时系统，实时系统处理的结果最终会由批处理系统来修正，正是这个观点使得 Storm 的设计与众不同。

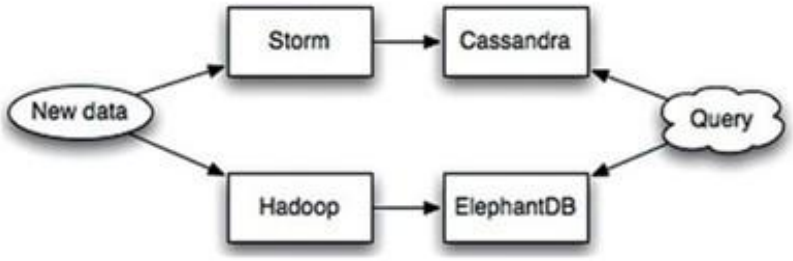

图 8-9 Twitter 数据系统分层处理架构

8.4.2Storm 主要特点

Storm 的主要特点如下：

- 简单的编程模型：Storm 降低了进行实时处理的复杂性；
- 支持各种编程语言：默认支持 Clojure、Java、Ruby 和 Python，要增加对其他语言的支持，只需实现一个简单的 Storm 通信协议即可；
- 容错性：Storm 会自动管理工作进程和节点的故障；
- 水平扩展：计算是在多个线程、进程和服务器之间并行进行的；
- 可靠的消息处理：Storm 保证每个消息至少能得到一次完整处理；
- 快速：系统的设计保证了消息能得到快速的处理；
- 本地模式：Storm 有一个“本地模式”，可以在处理过程中完全模拟 Storm 集群，这样可以快速进行开发和单元测试；
- 容易部署：Storm 集群易于部署，只需少量的安装和配置就可以运行。

Storm 的这些特点，特别是能可靠地处理消息，保证每条消息都能得到处理的特点，使其在目前的流计算应用中得到了广泛的使用。此外，Storm 支持本地模式，在单机上就可以进行安装、使用，大大降低了学习成本。

8.4.3Storm 应用领域

Twitter 列举了 Storm 的三大应用领域：

- 信息流处理（Stream Processing）：Storm 可以用来实时处理新数据和更新数据库，兼具容错性和可扩展性；
- 连续计算（Continuous Computation）：Storm 可以进行连续查询并把结果即时反馈给

客户，比如将 Twitter 上的热门话题发送到客户端；

- 分布式远程过程调用（Distributed RPC）：Storm 可以用来并行处理密集查询，Storm 的拓扑结构（后文会介绍）是一个等待调用信息的分布函数，当它收到一条调用信息后，会对查询进行计算，并返回查询结果。

除了这些领域，Storm 也可以应用于各类实时计算的应用场景。

8.4.4Storm 设计思想

Storm 对一些概念进行了抽象化，其主要术语和概念包括 Streams、Spouts、Bolts、Topology 和 Stream Groupings。

（1） Streams

如图 8-10 所示，在 Storm 对流 Stream 的抽象描述中，流是一个不间断的无界的连续 Tuple（元组，是元素有序列表）。这些无界的元组会以分布式的方式并行地创建和处理。

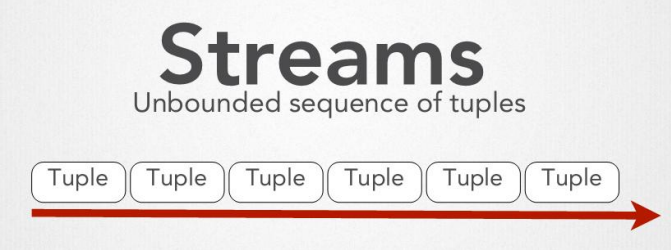

图 8-10 Streams:无界的 Tuples 序列

（2） Spouts

Storm 认为每个 Stream 都有一个源头，它将这个源头抽象为 Spouts。Spouts 会从外部读取流数据并发出 Tuple，如图 8-11 所示。

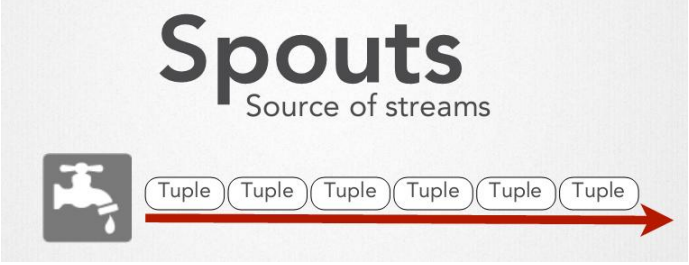

图 8-6 Spouts 数据源

（3） Bolts

如图 8-12 所示，Storm 将流的中间状态转换抽象为 Bolts，Bolts 可以处理 tuples，同时它也可以发送新的流给其他 Bolts 使用。Bolts 作为消息处理者，所有的消息处理逻辑被封装

在 Bolts 里面，处理输入的数据流并产生输出的新数据流。Bolts 中可执行过滤、聚合、查询数据库等操作。

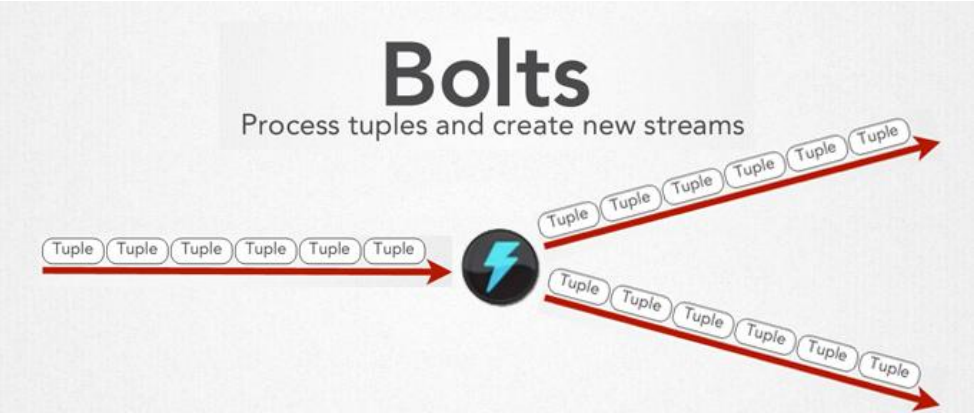

图 8-7 Bolts: 处理 tuples 并产生新的数据流

（4） Topology

为了提高效率，在 Spout 源可以接上多个 Bolts 处理器。Storm 将这样的无向环图抽象为 Topology，如图 8-13 所示。Topology 是 Storm 中最高层次的抽象概念，它可以被提交到 Storm 集群执行，一个拓扑就是一个流转换图。图中的边表示 Bolt 订阅了哪些流。当 Spout 或者 Bolt 发送元组到流时，它就发送元组到每个订阅了该流的 Bolt 上进行处理。

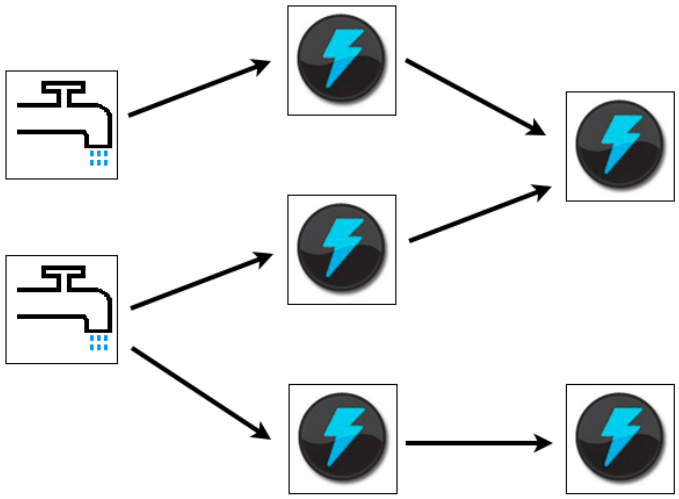

图 8-8 Topology 示意图

在 Topology 的实现上，Storm 中拓扑定义仅仅是一些 Thrift 结构体（Thrift 是基于二进制的高性能的通讯中间件），这样一来就可以使用其他语言来创建和提交拓扑。

Stream 中的每一个 Tuple 就是一个值列表。列表中的每个值都有一个名称，并且该值可以是基本类型、字符类型、字节数组等，当然也可以是其他可序列化的类型。

Topology 中的每个节点都要说明它所发射出的元组的字段的名称，这样其他节点只需

要订阅该名称就可以接收处理。

（5） Stream Groupings

消息分发策略，即定义一个 Stream 应该如何分配给 Bolts。目前 Stream Groupings 有如下几种方式：

- Shuffle Grouping：随机分组，随机分发 Stream 中的 Tuple；
- Fields Grouping：按字段分组，具有相同值的 Tuple 会被分发到对应的 Bolts；
- All Grouping：广播分发，每个 Tuple 都会被分发到各个 Bolts 中；
- Global Grouping：全局分组，Tuple 只会分发给 Bolt 中的一个任务；
- Non Grouping：不分组，与随机分组效果类似；
- Direct Grouping：直接分组，由 Tuple 的生产者来定义接收者。

通过这些消息分发策略，Storm 解决了两个组件（Spout 和 Bolt）之间如何发送 Tuple 的问题。

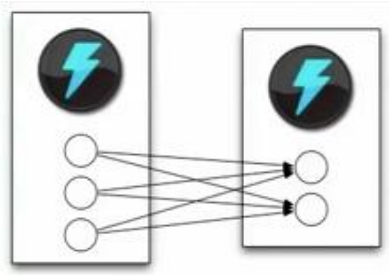

图 8-9 Stream grouping 示意图

图 8-14 中的箭头表示 Tuple 的流向，而圆圈则表示 Task，Task 就是具体的处理逻辑，每一个 Spout 和 Bolt 会被当作很多 Task 在整个集群里面执行，并且每一个 Task 对应到一个线程。通过一个完整的 Topology 示意图（如图 8-15 所示），可以了解 Stream Grouping 和 Task 在当中的作用。

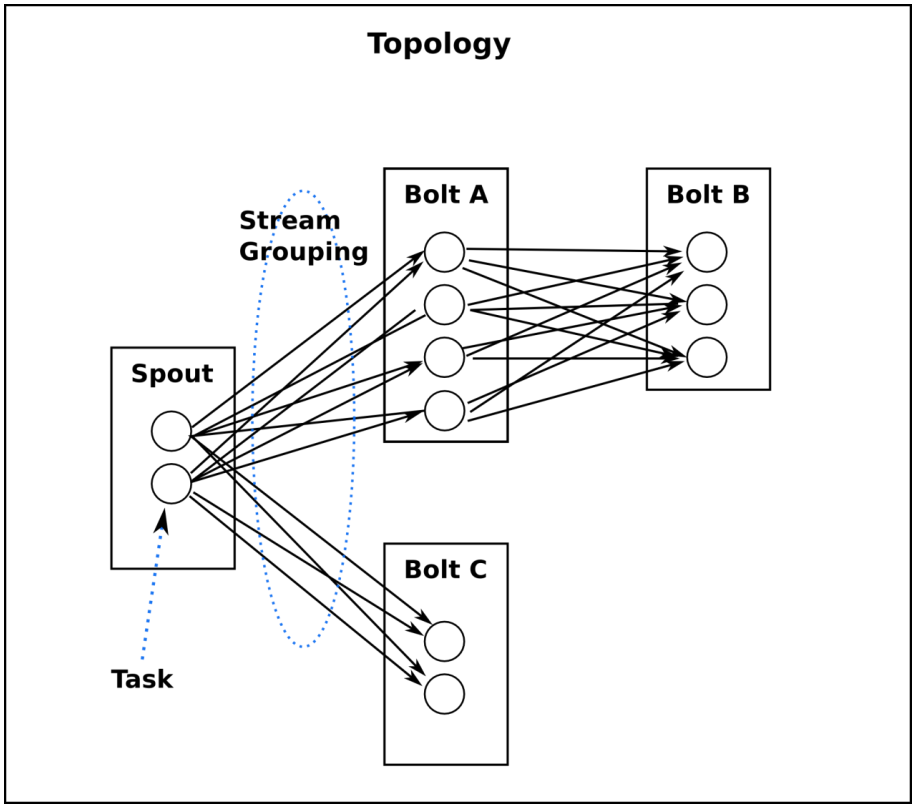

图 8-10 Topology 完整示意图

8.4.5Storm 框架设计

Storm 运行于集群之上，与 Hadoop 集群类似。但在 Hadoop 上运行的是“MapReduce Jobs”，而在 Storm 上运行的是“Topologies”。两者大不相同，一个关键不同是一个 MapReduce 的 Job 最终会结束，而一个 Topology 永远处理消息（或直到 kill 它）。

Storm 集群有两种节点：控制（Master）节点和工作者（Worker）节点。

Master 节点运行一个称之为“Nimbus”的后台程序，负责在集群范围内分发代码、为 worker 分配任务和故障监测。

每个 Worker 节点运行一个称之“Supervisor”的后台程序，监听分配给它所在机器的工作，基于 Nimbus 分配给它的事情来决定启动或停止工作者进程。

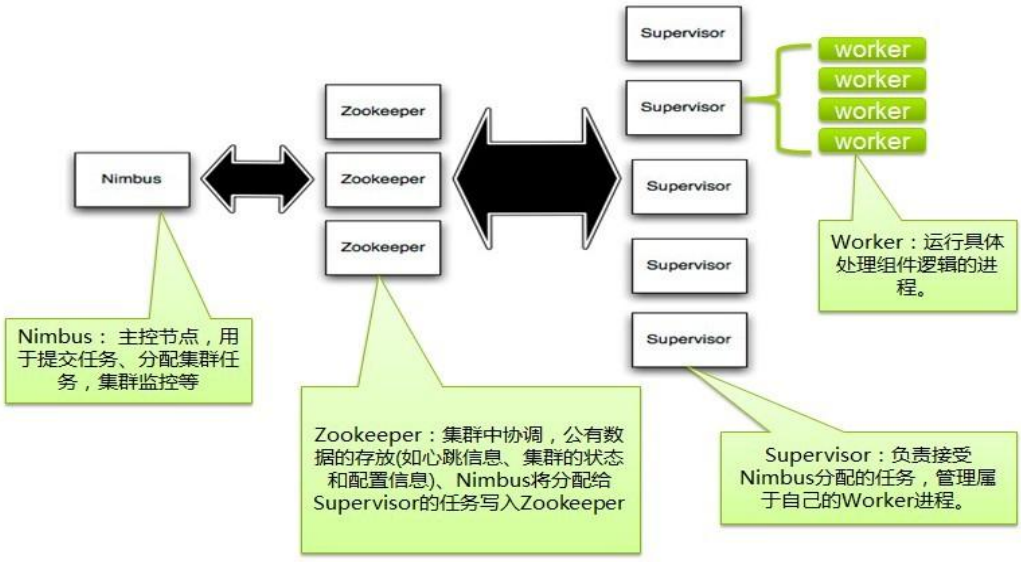

图 8-11 Storm 集群架构示意图

从图 8-16 可以看出，Storm 采用了 Zookeeper 来作为分布式协调组件，一个 Zookeeper 集群负责 Nimbus 和多个 Supervisor 之间的所有协调工作（一个完整的拓扑可能被分为多个子拓扑，并由多个 supervisor 完成）。

Nimbus 后台程序和 Supervisor 后台程序都是快速失败（fail-fast）和无状态的，所有状态维持在 Zookeeper 或本地磁盘中。

在这种设计中，master 节点并没有直接和 worker 节点通信，而是借助中介 Zookeeper，这样一来可以分离 master 和 worker 的依赖，将状态信息存放在 Zookeeper 集群内以快速回复任何失败的一方。

这意味着你可以 kill 杀掉 nimbus 进程和 supervisor 进程，然后重启，它们将恢复状态并继续工作，这种设计使得 Storm 极其稳定。

再来看看 Storm 的工作流程（如图 8-17 所示）：

- 首先定义 Topology，由客户端提交 Topology 到 Storm 中执行；
- Nimbus 建立 Topology 本地目录，将 Topology 分配到集群中进行处理（将分配给 Supervisor 的任务写入 Zookeeper 中）；
- Supervisor 从 Zookeeper 中获取所分配的任务，启动任务；
- Worker 节点中的 Task 执行具体的任务逻辑。

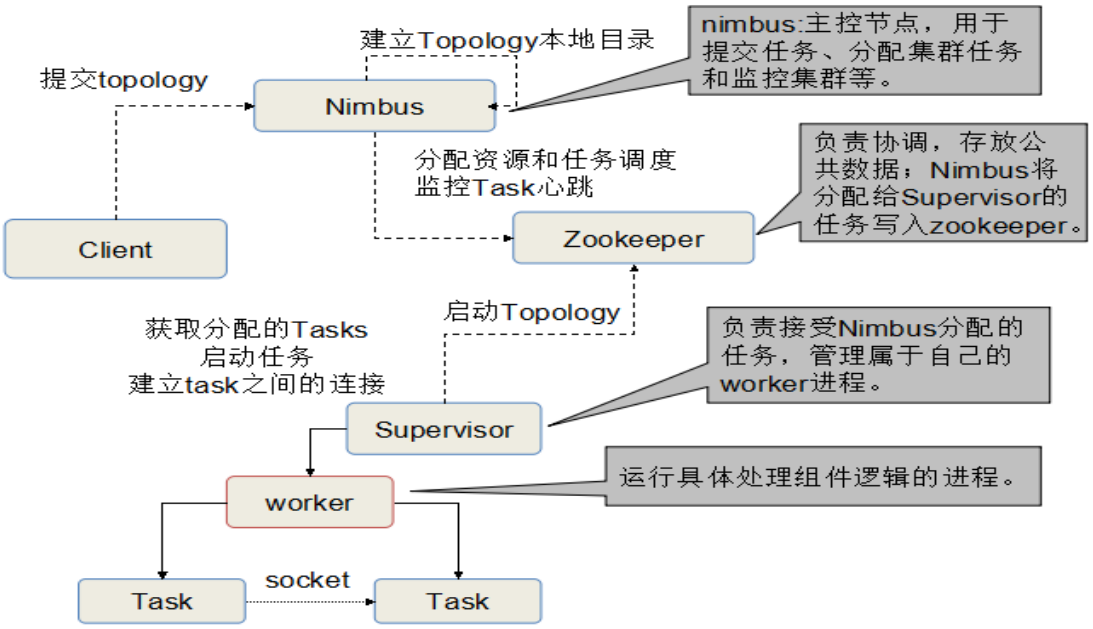

图 8-12 Storm 工作流程示意图

总的来说，流计算任务的整体逻辑在 Topology 中定义，然后便可提交到 Storm 中执行。

8.4.6Storm 实例

了解了 Storm 的设计思想和框架设计，下面以一个单词统计的实例来加深对 Topology 的认识。

Storm 编程模型非常简单，如下 Topology 代码即定义了整个单词统计的逻辑：

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("sentences", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8)
        .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 12)
        .fieldsGrouping("split", new Fields("word"));
```

图 8-13 单词统计 Topology 代码

代码中第一行新建了一个 Topology builder。

Builder.setSpout 是对 Spout 数据源的定义，方法中有三个参数，第一个参数定义 Spout 来源为“sentences”，表明要统计单词的来源；第二个参数定义 Spout 数据源的处理函数；参数三则定义了并发线程数。

紧接着代码包含两个 builder.setBolt 定义，同样有三个参数，并且每个 setBolt 同时定义

了消息分发策略。第一个 setBolt 定义了单词的分割，即从句子中提取出单词，并以随机分发的方式将 Tuple 分发给每个 Bolt。而第二个 setBolt 则定义了对这些分割后单词的处理，即计数，分发方式为“按字段分组”，只有具有相同 field 值的 Tuple 才会发给同一个 Task 进行统计，保证了统计的准确性。

从代码中也可以看出，Bolts 是通过订阅 Tuple 的名称来接收相应的数据，如第二个 setBolt 订阅了前一个 setBolt 分割后的单词数据。

Topology 中只是定义了整个计算逻辑，具体的处理函数则可以使用多种语言来完成。如 SplitSentence 方法中，代码 super(“python”, “splitsentence.py”)说明这个方法是使用 Python 语言来实现的。

```
public static class SplitSentence extends ShellBolt implements IRichBolt {

    public SplitSentence() {
        super("python", "splitsentence.py");
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }
}
```

图 8-19 SplitSentence 类定义

方法中调用了 splitsentence.py 脚本（如图 8-20 所示），该脚本定义了一个简单的单词分割方法，即通过空格来分割单词。当然真正的单词分割逻辑没有这么简单，这里仅是通过这个简单的实例代码来快速了解其实现原理。分割后的单词通过 emit 的方法将 Tuple 发射出去，以便订阅了该 Tuple 的 Bolts 进行接收。

```
import storm

class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
          storm.emit([word])
```

图 8-14 splitsentence.py 脚本

SplitSentence 类中的 declareOutputFields 方法定义了要输出的字段。进行”count”操作的 Bolts 接收其订阅的 Tuple 后，调用 WordCount 类来进行下一步的处理，如图 8-21 所示。

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if(count==null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

图 8-15 WordCout 类定义

图 8-21 的类定义中的 execute 方法说明了单词统计的逻辑，即单词若已统计过，则计数加 1，否则置为 0。同时 declareOutputFields 方法定义了最终的输出字段：(“word”, “count”)。

下图表示一个句子经过上面单词统计流程后的统计结果图。

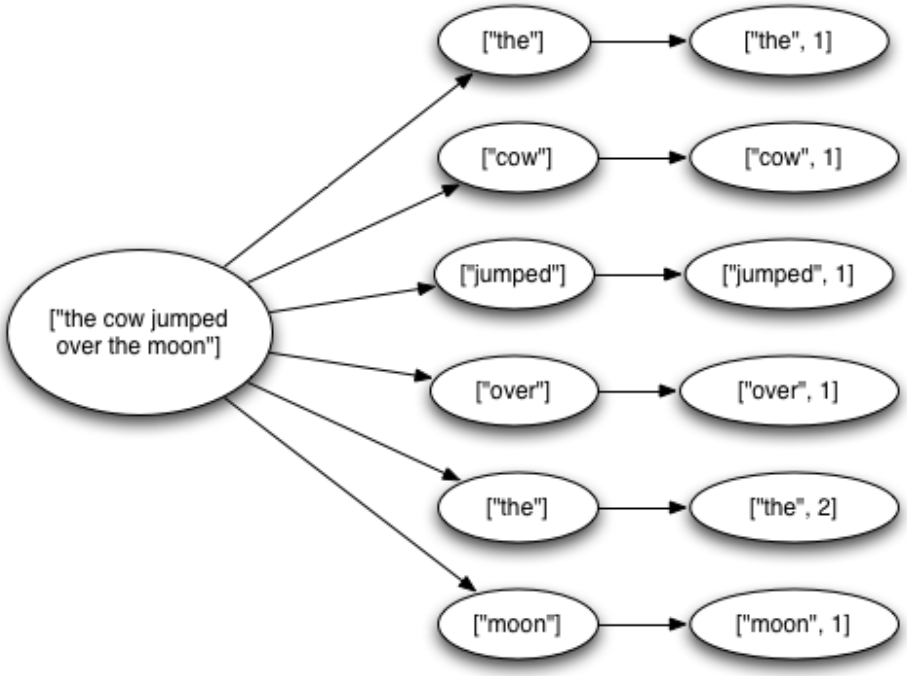

图 8-22 一个句子经单词统计后的统计结果示意图

现在让我们再来总结一下整个流程：

- 每个从 Spout 发送出来的消息（英文句子）都会触发很多的 Task 被创建；
- 用于分割单词的 Bolts 将句子分解为独立的单词，然后发射包含这些单词的 Tuple；

- 用于计数的 Bolts 接收 Tuple，并对其进行统计；
- 最后，实时的输出每个单词以及它出现过的次数。

这虽然是一个简单的单词统计，但对其进行扩展，便可应用在许多场景中，如微博中的实时热门话题。Twitter 也正是使用了 Storm 来实现这一功能。

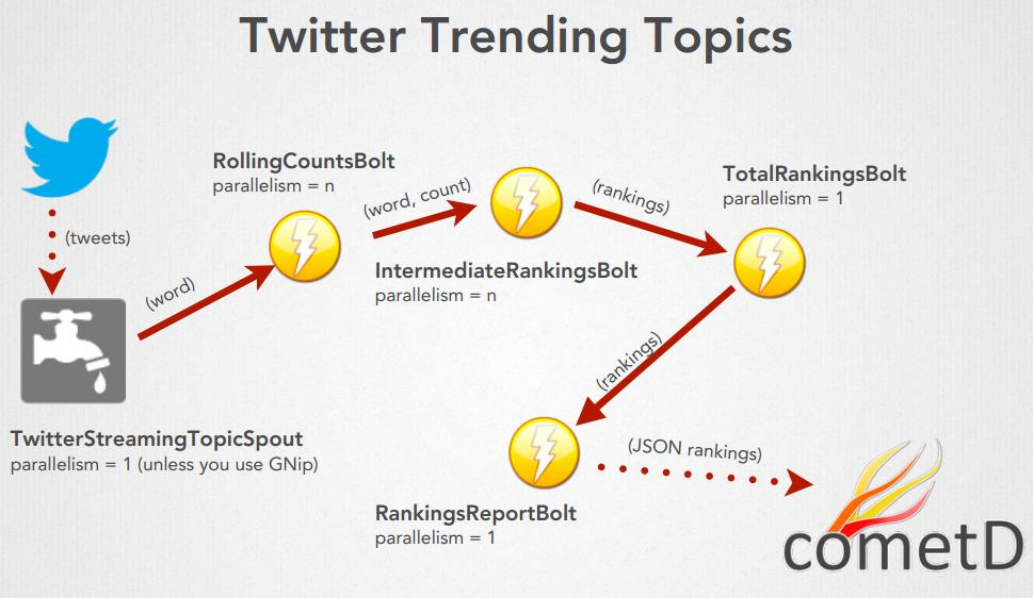

图 8-163 Twitter 实时热门话题处理流程示意图

Twitter 实时热门话题的处理流程与单词统计流程是相近的（如图 8-173 所示），不过 Twitter 实时热门话题使用了更多级的 Bolts。首先，以用户发布的 tweets 作为数据源，经过 TwitterStreamingTopicSpout 处理后，分发给 RollingCountsBolt（用于实现滑动窗口计数和 Top N 排序，网上有文章介绍了 Twitter 的这一 Rolling Count 算法：http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/ ）；然后，发出的（word，count）Tuple 再经过 IntermediateRankingsBolt 进行排序；最后，由 TotalRankingBolt 进行汇总得出总的热门话题排序。排序结果再交给 RankingsReportBolt 进行最后的处理，如进行筛选、提取、输出等。图 8-183 中的 cometD 是一项 Ajax 推送技术，表示处理后的排序结果以 JSON 的格式，结合 JavaScript 前端技术实时推送给 Web 前端展示给用户。

8.4.7 哪些公司在使用 Storm

Storm 自 2011 年发布以来，凭借其优良的实时流计算框架设计及开源特性，如今已经吸引了许多大型互联网公司的注意，并将其应用到了自身的实际项目中。下图展示了部分使

用 Storm 的公司和项目，典型用户如淘宝和阿里巴巴。

图 8-194 使用 Storm 的公司和项目

淘宝和阿里巴巴许多业务都需要实时流计算的支撑，如业务监控、广告推荐、买家实时数据分析等业务场景。淘宝数据部门开发的新架构已把 Storm 作为当中重要的一部分（如图 8-25 所示）。

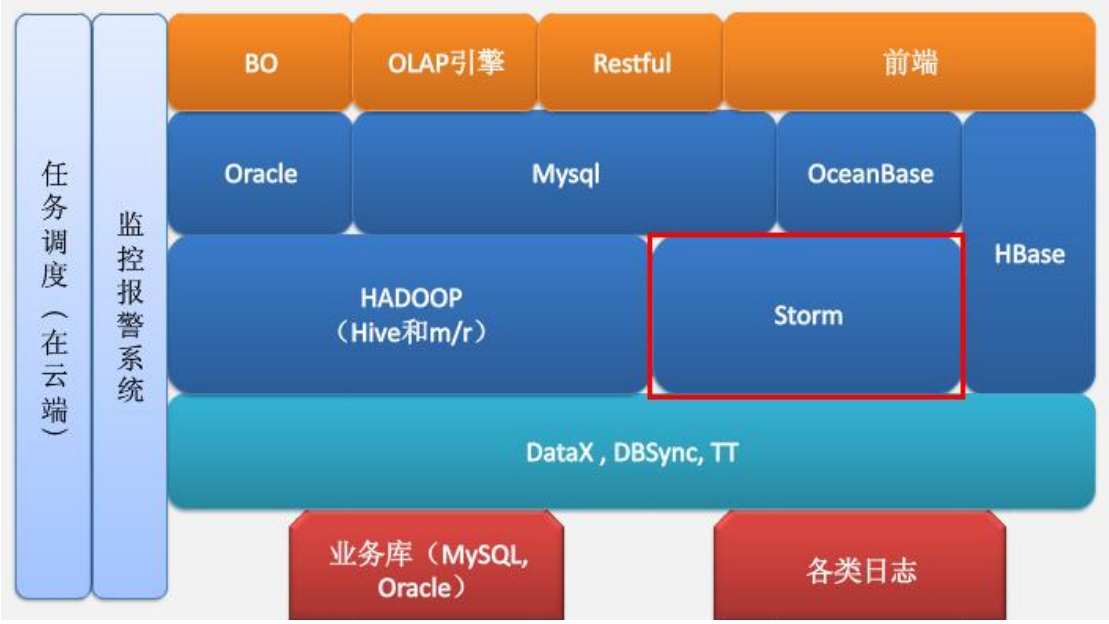

图 8-20 淘宝数据部门新架构示意图

8.4.8 流计算框架汇总

目前业内已涌现出许多的流计算框架与平台，在此做一个小小的汇总。

第一类是商业级的流计算平台，代表如下：

- IBM InfoSphere Streams：商业级高级计算平台，帮助用户开发的应用程序快速摄取、分析和关联来自数千个实时源的信息 http://www-03.ibm.com/software/products/cn/zh/infosphere-streams/；
- IBM StreamBase：IBM 开发的另一款商业流计算系统，在金融部门和政府部门使用 http://www.streambase.com/。

第二类是开源流计算框架，代表如下：

- Twitter Storm：免费、开源的分布式实时计算系统，可简单、高效、可靠地处理大量的流数据 http://storm-project.net/ ；
- Yahoo! S4（Simple Scalable Streaming System）：开源流计算平台，是通用的、分布式的、可扩展的、分区容错的、可插拔的流式系统。http://incubator.apache.org/s4/；

第三类是公司为支持自身业务开发的流计算框架，虽然未开源，但有不少的学习资料可供了解、学习，代表如下：

- Facebook Puma：Facebook 使用 Puma 和 HBase 相结合来处理实时数据；
- DStream：百度正在开发的属于百度的通用实时数据流计算系统；
- 银河流数据处理平台：淘宝开发的通用流数据实时计算系统；
- Super Mario：基于 Erlang 语言和 Zookeeper 模块开发的高性能数据流处理框架。

此外，业界也涌现出了像 SQLstream 这样专门致力于实时大数据流处理服务的公司。

本章小结

本章首先介绍了什么是流计算，介绍了流计算产生的背景与流计算的基本概念，接着介绍了流计算的处理模型与处理流程，分析了 Hadoop 为代表的批处理能否胜任流计算的工作。接着对流计算的应用场景做了总结，并通过具体的实例来说明当前流计算框架的重要性。接下来，着重介绍了目前流行的开源流计算框架 Twitter Storm，包括其主要特点、应用领域、设计思想和框架设计，并且通过一个简单的实例来加深对 Storm 的认知。最后对当前流计算框架做了一个小小的汇总。

参考文献

[1] Beyond MapReduce：谈 2011 年风靡的数据流计算系统 http://www.programmer.com.cn/9642/

[2] 对互联网海量数据实时计算的理解 http://www.cnblogs.com/panfeng412/archive/2011/10/28/2227195.html

[3] Storm - As deep into real-time data processing as you can get in 30 minutes. http://www.slideshare.net/DanLynn1/storm-as-deep-into-realtime-data-processing-as-you-can-get-in-30-minutes

[4] Storm 实时流计算 http://wenku.baidu.com/view/7b24d0d49e3143323968937d.html

实时流式数据处理及应用

http://365cy.gotoip1.com/wp-content/uploads/2013/06/05xuzhengjun.pdf

第 9 章 图计算

随着大数据时代的到来，图的规模越来越大，有的甚至有数十亿的顶点和数千亿的边，这就给高速地处理图数据带来了挑战。一台机器已经不能存放所有需要计算的数据了，所以需要一个分布式的计算环境。而已有的图计算框架和图算法库不能很好地满足计算需求，因此，新的图计算框架应运而生。

本章内容首先简单介绍了图计算，然后详细介绍了当前热门的 Google 图计算框架 Pregel，包括 Pregel 图计算模型、Pregel 中的 C++ API、Pregel 的执行过程和 Pregel 的算法实现，内容要点如下：

- 图计算简介
- Google Pregel 简介
- Google Pregel 图计算模型
- Pregel 的 C++ API
- Pregel 模型的基本体系结构
- Pregel 模型的应用实例
- 改进的图计算模型

9.1 图计算简介

在实际应用中，存在许多图计算问题，比如最短路径、集群、网页排名、最小切割、连通分支等等。图计算算法的性能，直接关系到应用问题解决的高效性，尤其对于大型图（比如社交网络和网络图）而言，更是如此。下面我们首先指出传统图计算解决方案的不足之处，然后介绍两大类通用图计算软件。

9.1.1 传统图计算解决方案的不足之处

在很长一段时间内，都缺少一个可扩展的通用系统来解决大型图的计算问题。很多传统的图计算算法都存在以下几个典型问题：（1）常常表现出比较差的内存访问局部性；（2）针对单个顶点的处理工作过少；（3）计算过程中伴随着并行度的改变。

针对大型图（比如社交网络和网络图）的计算问题，可能的解决方案及其不足之处具体

如下：

- 为特定的图应用定制相应的分布式实现。不足之处是，在面对新的图算法或者图表示方式时，就需要做大量的重复实现，不通用。
- 基于现有的分布式计算平台进行图计算。但是，在这种情况下，它们往往并不适于做图处理。比如，MapReduce 就是一个对许多大规模计算问题都非常合适的计算框架。有时，它也被用来对大规模图对象进行挖掘，但是，通常在性能和易用性上都不是最优的。尽管这种对数据处理的基本模式经过扩展，已经可以使用方便的聚合以及类似于 SQL 的查询方式，但是，这些扩展对于图算法这种更适合用消息传递模型的问题来说，通常并不理想。
- 使用单机的图算法库。比如 BGL、LEAD、NetworkX、JDSL、Standford GraphBase 和 FGL 等等；但是，这种方式对可以解决的问题的规模提出了很大的限制。
- 使用已有的并行图计算系统。Parallel BGL 和 CGMgraph 这些库实现了很多并行图算法，但是，并没有解决对大规模分布式系统中来说非常重要的容错等一些问题。

9.1.2 图计算通用软件

正是因为传统的图计算解决方案无法解决大型图的计算问题，因此，就需要设计能够用来解决这些问题的通用图计算软件。针对大型图的计算，目前通用的图处理软件主要包括两种：第一种主要是基于遍历算法和实时的图数据库，如 Neo4j 、OrientDB、DEX 和 InfiniteGraph。第二种则是以图顶点为中心的消息传递批处理的并行引擎，如 Hama、Golden Orb、Giraph 和 Pregel。

第一种图处理软件，基本都基于 Tinkerpop 的图基础框架，Tinkerpop 项目关系如图 9-1 所示。

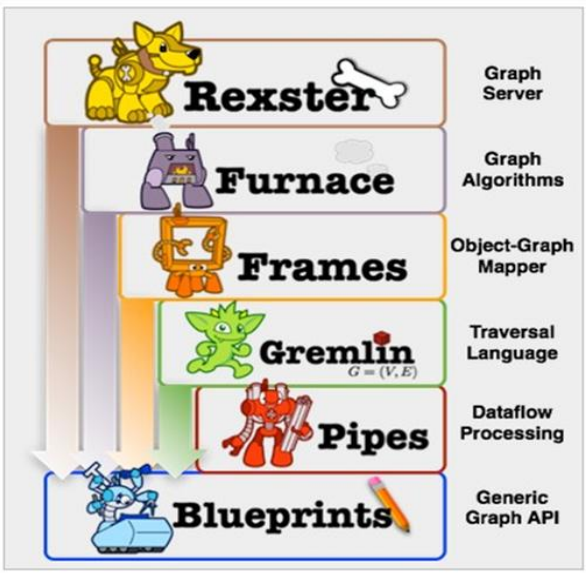

图 9-21 Tinkerpop 项目关系图

下面具体介绍一下 TinkerPop 框架的各层功能：

- **Blueprints**：是一组针对属性图数据模型的接口、实现、测试套件，它和 JDBC 类似，但是，它是基于图数据库的。就其本身而言，它提供了一组通用的接口，允许开发者对其图数据库后台即插即用。另外，在 Blueprints 上编写的软件可以运行于所有的 Blueprints 开启的图数据库。在 Tinkerpop 的图基础框架中，Blueprints 为其他几层提供基础技术服务。
- **Pipes**：是一个应用流程图的数据流框架。一个流程图由很多通过“通信边”相连的 pipe 顶点组成。一个 pipe 实现了一个简单的计算步骤，它可以和其他的 pipe 相组合，一起产生一个更大的计算。这样的数据流图允许拆分、合并、循环和输入输出数据的相互转换。伴随着主 Pipe 的分布，会产生大量的 Pipe 类。只要了解了每个 Pipe 的实现，就可以直接使用 Pipe 框架。
- **Gremlin**：是一种图遍历语言，可以用于图的查询、分析和处理。它工作在那些执行 Blueprints 性能图数据模型的图数据库和框架上。Gremlin 是能用于各种 JVM 语言的一种图遍历。Gremlin 的布局为 Java 和 Groovy 提供了支持。
- **Frames**：把 Blueprints 图映射为一组相互关联的域对象集。Frames 中经常使用 InvocationHandler、Porxy 类和 Annotations，使开发者可以用一个特殊的 Java 接口来构造一个图元素（顶点或边）。通过 Frames，非常容易确认图中数据各自的图解，对应哪一个带有注释的 Java 接口。
- **Furnace**：是能启用 Blueprints 图的一个算法包。在图理论和分析的历史进程中开发

了很多的图算法。这些算法中的大多数是为无标号的、单一关系的图而设计的。Furnace 的目的就是在单一关系图算法中揭示属性图（像属性化图或多关系图）。另外，Furnace 提供了针对不同图计算场景，各种图算法的优化实现，像单机图和分布图。

- **Rexster**：是一个图服务器，通过 REST 和一个被称为 RexPro 的二进制协议展现任意的 Blueprints 图。HTTP 网页服务器提供了标准的低层 GET、POST、PUT 和 DELETE 方法，一个灵活的、像开发一个外部服务器（如通过 Gremlin 的特殊图查询）一样允许插件法的扩展模型，用 Gremlin 编写的服务器端存储程序和一个基于浏览器的接口——Dog House。Rexster Console 使对 Rexster 服务器中的配置图的远程脚本评估成为可能。Rexster Kibbles 是由 TinkerPop 提供的各种 Rexster 服务器的扩展集。

第二种图处理软件，则主要是基于 BSP 模型所实现的并行图处理包。BSP 是由哈佛大学 Viliant 和牛津大学 Bill McColl 提出的并行计算模型，全称为“整体同步并行计算模型”(Bulk Synchronous Parallel Computing Model,简称 BSP 模型),又名“大同步模型”。创始人希望 BSP 模型像冯·诺伊曼体系结构那样，架起计算机程序语言和体系结构间的桥梁，故又称作“桥模型” (Bridge Model)。一个 BSP 模型由大量相互关联的处理器所组成，它们之间形成了一个通信网络。每个处理器都有快速的本地内存和不同的计算线程。一次 BSP 计算过程包括一系列全局超步，所谓的超步就是计算中的一次迭代。每个超步主要包括三个组件：

- **并发计算**：每个参与的处理器都有自身的计算任务，它们只读取存储在本地内存的值，这些计算都是异步并且独立的；
- **通讯**：处理器群相互交换数据，交换的形式是，由一方发起推送(put)和获取(get)操作；
- **栅栏同步**(Barrier synchronisation): 当一个处理器遇到“路障”，会等到其他所有处理器完成它们的计算步骤；每一次同步也是一个超步的完成和下一个超步的开始；图 9-2 是一个超步的垂直结构图。

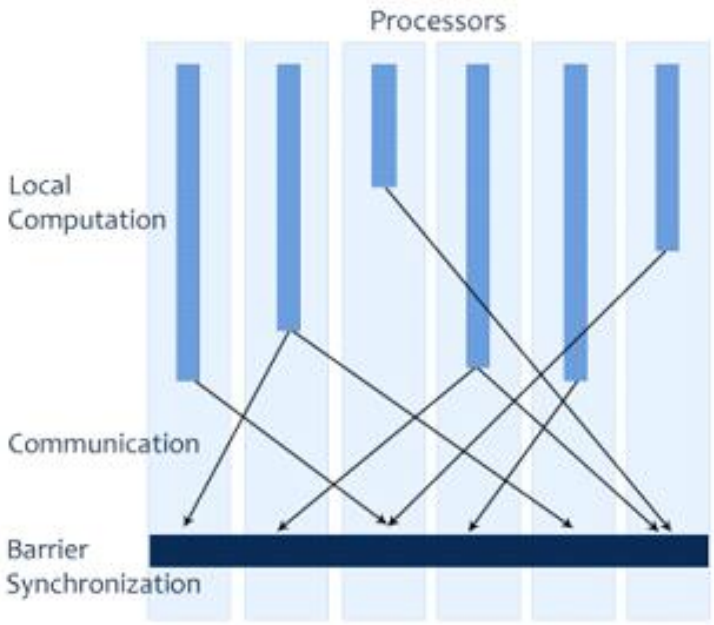

图 9-2 一个超步的垂直结构图

9.2 Google Pregel 简介

Pregel就是一种基于BSP模型所实现的并行图处理包。为了解决大型图的分布式计算问题，Pregel搭建了一套可扩展的、有容错机制的平台，该平台提供了一套非常灵活的API，可以描述各种各样的图计算。Pregel是一个用于分布式图计算的计算框架，主要用于图遍历（BFS）、最短路径（SSSP）、PageRank计算等等。共享内存的运行库有很多，但是，对于Google来说，一台机器早已经放不下需要计算的数据了，所以，需要分布式这样一个计算环境。

没有Pregel之前，你可以选择用MapReduce来做，但是效率很低。图算法如果用MapReduce实现，需要一系列的MapReduce的调用。从一个阶段到下一个阶段，它需要传递整个图的状态，会产生大量不必要的序列化和反序列化开销。而Pregel使用超步简化了这个过程。下面我们以一个实例来阐述采用Pregel和MapReduce来执行图计算的区别。

实例：PageRank算法在Pregel和MapReduce中的实现。

PageRank 算法作为 Google 的网页链接排名算法，具体公式如下：

$$PR = \beta \sum_{i-1}^{n} \frac{PR_i}{N_i} + (1-\beta)\frac{1}{N}$$

对于任意一个链接，其PR值为链入到该链接的源链接的PR值对该链接的贡献和（分母

Ni为第i个源链接的链出度）。

Pregel是Google提出的专门为图计算所设计的计算模型，主要来源于BSP并行计算模型的启发。要用Pregel计算模型实现PageRank算法，也就是将网页排名算法映射到图计算中，这其实是很自然的，因为，网络链接是一个连通图。

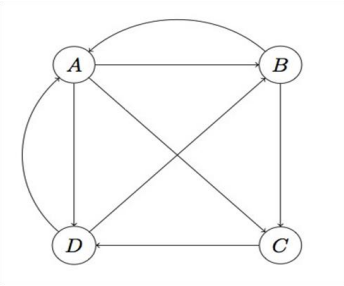

图 9-3 一个连通图

图9-3就是四个网页（A,B,C,D）互相链入链出组成的连通图。根据Pregel的计算模型，将计算定义到顶点（即A，B，C，D）上来，每个顶点对应一个对象，即一个计算单元。

每一个计算单元包含三个成员变量：

- Vertex value：顶点对应的PR值；
- Out edge：只需要表示一条边，可以不取值；
- Message：传递的消息，因为需要将本顶点对其它顶点的PR贡献值，传递给目标顶点。

每一个计算单元包含一个成员函数：

- Compute：该函数定义了顶点上的运算，包括该顶点的PR值计算，以及从该顶点发送消息到其链出顶点。

PageRank算法的Pregel实现代码如下：

```
class PageRankVertex
    : public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
            sum += msgs->Value();
            *MutableValue() =
                0.15 / NumVertices() + 0.85 * sum;
        }
        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
```

```
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
};
```

PageRankVertex 继承自 Vertex 类。顶点 value 类型是 double，用来保存 PageRank 中间值，消息类型也是 double，用来传输 PageRank 分数，边的 value 类型是 void，因为不需要存储任何信息。我们假设，在第 0 个超步时，图中各顶点的 value 值被初始化为 1/NumVertices()。在前 30 个超步中，每个顶点都会沿着它的出射边，发送它的 PageRank 值除以出射边数目以后的结果值。从第 1 个超步开始，每个顶点会将到达的消息中的值加到 sum 值中，同时将它的 PageRank 值设为 0.15/ NumVertices()+0.85*sum。到了第 30 个超步后，就没有需要发送的消息了，同时所有的顶点 VoteToHalt。在实际中，PageRank 算法需要一直运行直到收敛，可以使用 aggregators 来检查是否满足收敛条件。

MapReduce 也是 Google 提出的一种计算模型，它是为全量计算而设计。采用 MapReduce 实现 PageRank 的计算过程需要包括以下三个阶段：

- **阶段 1：解析网页**

Map task 把（URL，page content）映射为（URL，（PRinit，list-of-urls）），其中，PRinit 是 URL 的“seed”PageRank，list-of-urls 包含了该 URL 页面中的外链所指向的所有页。Reduce task 只是恒等函数。

在阶段 1 中，对于 MapReduce 程序，Map 函数的输入是用户指定的文件（这里就是一个网页），那么，输入的 value 值就是网页内容，key 值是网页的 url。程序对输入的 key 和 value 进行一系列操作，然后，按(key,value)键值对的形式输出。系统获取 Map 函数的输出，把相同的 key 合并，再把 key 和 value 集合作为键值对作为 reduce 函数的输入。程序员自定义 reduce 函数中的处理方法（这里是一个恒等函数），输出(key,value)键值对到磁盘文件（这里的键值对中的 key 仍是该网页的 url，value 是该页的初始 PR 值和链出页的 url 列表）。

- **阶段 2：PageRank 分配**

Map task 得到（URL，（cur_rank，url_list）），对于每一个 url_list 中的 u，输出（u，cur_rank/|url_list|），并输出链接关系（URL，url_list），用于迭代。

Reduce task 获得（URL，url_list）和很多（URL，var）值对，对于具有相同 key 值的

value 进行汇总，并把汇总结果乘以 d(这里是 0.85)，然后输出输出（URL，（new_rank，url_list））。

阶段 2 是一个迭代过程，每次将磁盘上的文件读入，key 为每一个网页链接，value 的第一项是当前网页的 PR 值，第二项是该网页中的外链列表。Map 函数将输入的每一对(key,value)“反转”成多对(value,key)输出，也就是说，每个输出的 key 为输入 value 中的每一个外链，而输出的 value 则为输入 key 的链接、输入 key 的 PR 值除以输入 key 的对外链接数。

在 Reduce 函数中，就可以根据输入的 value 中的参数来计算每一个 key 新的 PageRank 值，并按 Map 函数的格式输出到磁盘，以作为下一次迭代的输入。迭代多次后，PageRank 值趋于稳定，就得出了较为精确的 PageRank 值。

- **阶段 3：最后阶段**

一个非并行组件决定是否达到收敛。如果达到收敛，写出 PageRank 生成的列表。否则，回退到第 2 阶段的输出，进行另一个第 2 阶段的迭代。

下面具体解释一下阶段 2 的伪代码实现：

Mapper 函数的伪码：

```
input <PageN, RankN> -> PageA, PageB, PageC ... // 链接关系
begin
    Nn := the number of outlinks for PageN;
    for each outlink PageK
        output PageK -> <PageN, RankN/Nn>
    // 同时输出链接关系，用于迭代
    output PageN -> PageA, PageB, PageC ...
end
```

Mapper 的输出如下（已经排序，所以 PageK 的数据排在一起，最后一列则是链接关系对）：

```
PageK -> <PageN1, RankN1/Nn1>
PageK -> <PageN2, RankN2/Nn2>
...
PageK -> <PageAk, PageBk, PageCk>
```

Reduce 函数的伪码：

```
input mapper's output
begin
    RankK := 0;
    for each inlink PageNi
        RankK += RankNi/Nni * beta
    // output the PageK and its new Rank for the next iteration
    output <PageK, RankK> -> <PageAk, PageBk, PageCk...>
end
```

上述伪代码只是一次迭代的代码，多次迭代需要重复运行，需要说明的是这里可以优化的地方很多，比如把 Mapper 的<pageN,pageA-pageB-pageC…>内容缓存起来，这样就不用再 output 作为 Reducer 的 input 了，同时，Reducer 在 output 的时候也不用传递同样的 <pagek,pageO-pageP-pageQ>，减少了大量的 I/O，因为，在 PageRank 计算是，类似这样的数据才是主要数据。

简单地来讲，Pregel 将 PageRank 处理对象看成是连通图，而 MapReduce 则将其看成是 Key-Value 对。Pregel 将计算细化到顶点 vertex，同时在 vertex 内控制循环迭代次数，而 MapReduce 则将计算批量化处理，按任务进行循环迭代控制。图算法如果用 MapReduce 实现，需要一系列的 MapReduce 的调用。从一个阶段到下一个阶段，它需要传递整个图的状态，会产生大量不必要的序列化和反序列化开销。而 Pregel 使用超步简化了这个过程。

9.3 Google Pregel 图计算模型

9.3.1 Pregel 的消息传递模型

Pregel 选择了一种纯消息传递模型（如图 9-4 所示），忽略远程数据读取和其他共享内存的方式。这样做的原因有两个：第一，消息传递有足够的表达能力，没必要使用远程读取，还没有发现哪种算法是消息传递所不能表达的；第二是出于性能的考虑，在一个集群环境中，从远程机器上读取一个值是会有很高的延迟的，这种情况很难避免，而 Pregel 的消息传递模式通过异步和批量的方式传递消息，可以缓解这种远程读取的延迟。

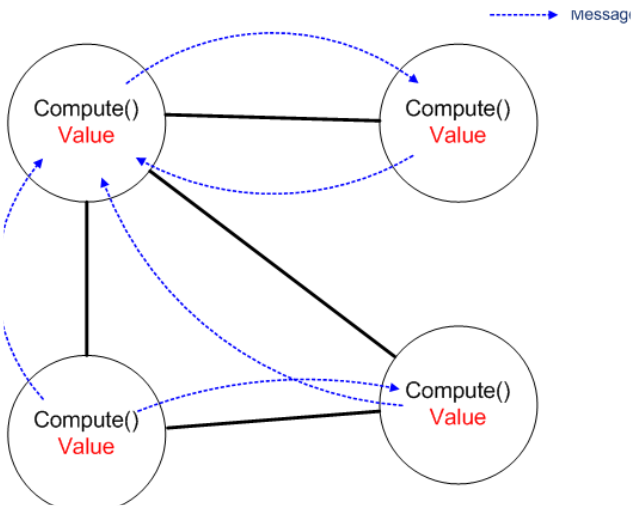

图 9-22 纯消息传递模型图

9.3.2 Pregel 的计算过程

Pregel 的计算过程是由一系列被称为超步(superstep)的迭代(iterations)组成的。在每一个超步中，计算框架都会针对每个顶点调用用户自定义的函数，这个过程是并行的。该函数描述的是一个顶点 V 在一个超步 S 中需要执行的操作。该函数可以读取前一个超步(S-1)中发送给 V 的消息，并发送消息给其他顶点，这些消息将会在下一个超步(S+1)中被接收，并且在此过程中修改顶点 V 及其出射边的状态。消息通常沿着顶点的出射边发送，但一个消息可能会被发送到任意已知 ID 的顶点上去。

9.3.3 Pregel 计算模型的实体

在 Pregel 计算模型中，输入是一个有向图，该有向图的每一个顶点都有一个相应的由 String 描述的顶点标识符。每一个顶点都有一个与之对应的可修改的用户自定义值。每一条有向边都和其源顶点关联，并且也拥有一个可修改的用户自定义值，并同时记录了其目标顶点的标识符。

在每个超步中，顶点的计算都是并行的，每个顶点执行相同的用于表达给定算法逻辑的用户自定义函数。每个顶点可以修改其自身及其出射边的状态，接收前一个超步(S-1)中发送给它的消息，并发送消息给其他顶点(这些消息将会在下一个超步中被接收)，甚至是修改整个图的拓扑结构。在这种计算模式中，“边”并不是核心对象，没有相应的计算运行在其上。

9.3.4 Pregel 计算模型的进程

算法是否能够结束，取决于是否所有的顶点都已经“vote”标识其自身已经达到“halt”状态了。在第 0 个超步，所有顶点都处于 active 状态，所有的 active 顶点都会参与所有对应超步中的计算。顶点通过将其自身的 status 设置成“halt”来表示它已经不再 active。这就表示该顶点没有进一步的计算需要去执行，除非再次被外部触发，而 Pregel 框架将不会在接下来的超步中执行该顶点，除非该顶点收到其它顶点传送的消息。如果顶点接收到消息被唤醒进入 active 状态，那么在随后的计算中该顶点必须显式地“deactive”。整个计算在所有顶点都达到“inactive”状态，并且没有 message 在传送的时候才宣告结束。这种简单的状态机如图 9-5 所示。

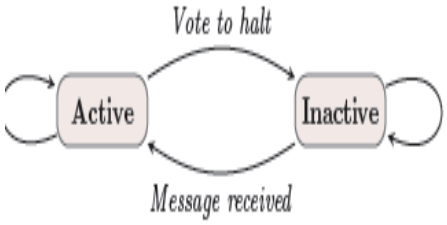

图 9-5 一个简单的状态机图

下面通过一个简单的例子来说明这些基本概念。

给定一个强连通图（如图 9-6 所示），图中每个顶点都包含一个值，它会将最大值传播到每个顶点。在每个超步中，顶点会从接收到的消息中选出一个最大值，并将这个值传送给其所有的相邻顶点。当某个超步中已经没有顶点更新其值，那么算法就宣告结束。

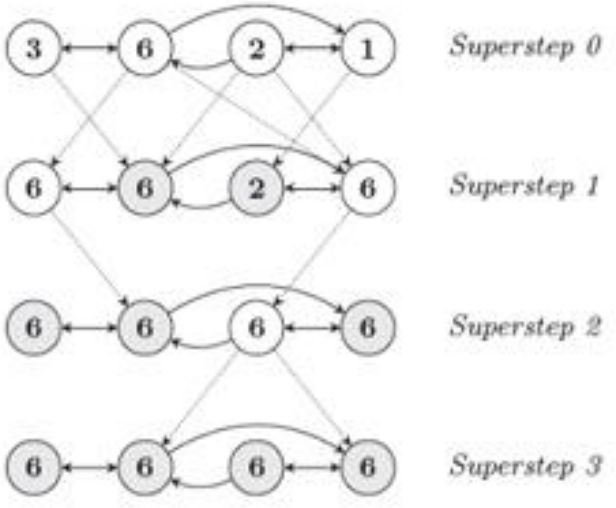

图 9-6 一个求最大值的步骤图

9.4 Pregel 的 C++ API

编写一个 Pregel 程序需要继承 Pregel 中已预定义好的一个基类——Vertex 类，如下所示：

```
template <typename VertexValue, typename EdgeValue, typename MessageValue>
class Vertex {
  public:
    virtual void Compute(MessageIterator* msgs) = 0;
    const string& vertex_id() const;
    int64 superstep() const;
    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();

    void SendMessageTo(const string& dest_vertex,
                       const MessageValue& message);
    void VoteToHalt();
};
```

该类的模板参数中定义了三个值类型参数，分别表示顶点、边和消息。每一个顶点都有一个对应的给定类型的值。这种形式可能看上有很多限制，但用户可以用 protocol buffer 来管理增加的其他定义和属性。而边和消息类型的行为比较类似。

用户覆写 Vertex 类的虚函数 Compute()，该函数会在每一个超步中对每一个顶点进行调用。预定义的 Vertex 类方法允许 Compute()方法查询当前顶点及其边的信息，以及发送消息到其他的顶点。Compute()方法可以通过调用 GetValue()方法来得到当前顶点的值，或者通过调用 MutableValue()方法来修改当前顶点的值。同时还可以通过由出射边的迭代器提供的方法来查看修改出射边对应的值。这种状态的修改是立时可见的。由于这种可见性仅限于被修改的那个顶点，所以，不同顶点并发进行的数据访问是不存在竞争关系的。

顶点和其对应的边所关联的值，是唯一需要在超步之间持久化的顶点级状态。将由计算框架管理的图状态限制在一个单一的顶点值或边值的这种做法，简化了主计算流程、图的分布以及故障恢复。

9.4.1 消息传递机制

顶点之间的通信是直接通过发送消息来实现的，每条消息都包含了消息值和目标顶点的名称。消息值的数据类型是由用户通过 Vertex 类的模版参数来指定。

在一个超步中，一个顶点可以发送任意多的消息。当顶点 V 的 Compute()方法在 S+1 超步中被调用时，所有在 S 超步中发送给顶点 V 的消息都可以通过一个迭代器来访问到。在该迭代器中并不保证消息的顺序，但是，可以保证消息一定会被传送并且不会重复。

一种通用的使用方式为：对一个顶点 V，遍历其自身的出射边，向每条出射边发送消息到该边的目标顶点，如 PageRank 算法所示的那样。但是，消息要发送到的目标顶点 dest_vertex，并不一定是顶点 V 的相邻顶点。一个顶点可以从之前收到的消息中获取到其非相邻顶点的标识符，或者顶点标识符可以隐式地得到。比如，图可能是一个 clique(一个图中两两相邻的一个点集，或是一个完全子图)，顶点的命名规则都是已知的(从 V1 到 Vn)，在这种情况下甚至都不需要显式地保存边的信息。

当任意一个消息的目标顶点不存在时，便执行用户自定义的 handlers。比如在这种情况下，一个 handler 可以创建该不存在的顶点或从源顶点中删除这条边。

找最大值的代码实现如下：

```
Class MaxFindVertex
        : public Vertex<double, void, double> {
    public:
        virtual void Compute(MessageIterator* msgs) {
            int currMax = GetValue();
            SendMessageToAllNeighbors(currMax);
            for ( ; !msgs->Done(); msgs->Next()) {
                if (msgs->Value() > currMax)
                    currMax = msgs->Value();
            }
            if (currMax > GetValue())
                *MutableValue() = currMax;
            else VoteToHalt();
```

```
        }
};
```

9.4.2 Combiner

发送消息，尤其是当目标顶点在另外一台机器上时，会产生一些开销。某些情况可以在用户的协助下降低这种开销。比方说，假如 Compute() 收到许多的 int 值消息，而它仅仅关心的是这些值的和，而不是每一个 int 的值，这种情况下，系统可以将发往同一个顶点的多个消息合并成一个消息，该消息中仅包含它们的“和值”，这样就可以减少传输和缓存的开销。

Combiners 在默认情况下并没有被开启，这是因为要找到一种对所有顶点的 Compute() 函数都合适的 Combiner 是不可能的。而用户如果想要开启 Combiner 的功能，需要继承 Combiner 类，覆写其 virtual 函数 Combine()。框架并不会确保哪些消息会被 Combine 而哪些不会，也不会确保传送给 Combine()的值和 Combining 操作的执行顺序。所以，Combiner 只应该对那些满足交换律和结合律的操作才给予打开。

对于某些算法来说，比如单源最短路径，我们观察到通过使用 Combiner 可以把流量降低 4 倍多。

下面是有关 combiner 应用的例子。假设我们想统计在一组相关联的页面中所有页面的链接数。在第一个迭代中，对从每一个顶点（页面）出发的链接，我们会向目标页面发送一个消息。这里输入消息上的 count 函数可以通过一个 combiner 来优化性能。在上面求最大值的例子中，一个 Max combiner 可以减少通信负荷（如图 9-7 所示），比如，假设顶点 1 和 6 在一台机器上，顶点 2 和 3 在另一台机器上，顶点 3 向顶点 6 传递的值是 3，顶点 2 向顶点 6 传递的值是 2，顶点 2 和顶点 3 都需要把消息传递到目标顶点 6，因此，可以采用 Combine() 函数把这两个消息进行合并后再发送给目标顶点 6，这样就减少了网络通信负责。

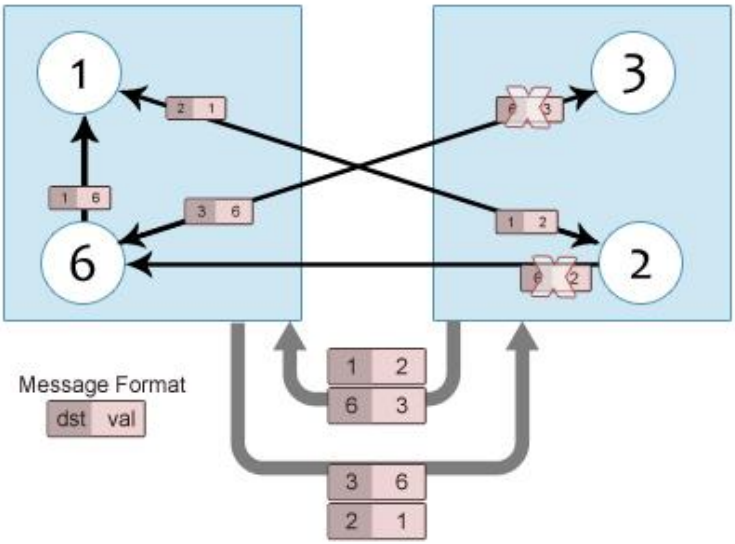

图 9-7 combiner 应用的例子

9.4.3 Aggregator

Pregel 的 Aggregators 是一种提供全局通信、监控和数据查看的机制。在一个超步 S 中，每一个顶点都可以向一个 Aggregator 提供一个数据，系统会使用一种 reduce 操作来负责聚合这些值，而产生的值将会对所有的顶点在超步 S+1 中可见。Pregel 包含了一些预定义的 aggregators，比如可以在各种整数和 string 类型上执行的 min、max、sum 操作。

Aggregators 可以用来做统计。例如，一个 sum aggregator 可以用来统计每个顶点的出度，最后相加就是整个图的边的条数。更复杂的一些 reduce 操作还可以产生统计直方图。

Aggregators 也可以用来做全局协同。例如，Compute()函数的一些逻辑分支可能在某些超步中执行，直到当“and” aggregator 表明所有顶点都满足了某条件时，会去执行另外的逻辑分支直到结束。又比如一个作用在顶点 ID 之上的 min 和 max aggregator，可以用来选定某顶点在整个计算过程中扮演某种角色等。

要定义一个新的 aggregator，用户需要继承预定义的 Aggregator 类，并定义在第一次接收到输入值后如何初始化，以及如何将接收到的多个值最后 reduce 成一个值。Aggregator 操作也应该满足交换律和结合律。

默认情况下，一个 aggregator 仅仅会对来自同一个超步的输入进行聚合，但是，有时也可能需要定义一个 sticky aggregator，它可以从所有的超步中接收数据。这是非常有用的，比如要维护全局的边条数，那么就仅仅在增加和删除边的时候才调整这个值了。

还可以有更高级的用法。比如，可以用来实现一个△-stepping 最短路径算法所需要的分

布式优先队列。每个顶点会根据它的当前距离分配一个优先级 bucket。在每个超步中，顶点将它们的 indices 汇报给 min aggregator。在下一个超步中，将最小值广播给所有 worker，然后让在最小 index 的 bucket 中的顶点放松它们的边。

一个有关 aggregator 应用的例子：Sum 运算符应用于每个顶点的出射边数，可以用来生成图中边的总数并使它能与所有的顶点相通信。

更复杂的规约运算符甚至可以产生直方图。在求最大值得例子中，我们可以通过运用一个 Max aggregator 在一个超步中完成整个程序。

9.4.4 Topology mutation

有一些图算法可能需要改变图的整个拓扑结构。比如一个聚类算法，可能会将每个聚类替换成一个单一顶点，又比如一个最小生成树算法，会删除所有除了组成树的边之外的其他边。正如用户可以在自定义的 Compute()函数能发送消息，同样可以产生在图中增添和删除边或顶点的请求。

多个顶点有可能会在同一个超步中产生冲突的请求(比如两个请求都要增加一个顶点 V，但初始值不一样)。Pregel 中用两种机制来决定如何调用：局部有序和 handlers。

由于是通过消息发送的，拓扑改变在请求发出以后，在超步中可以高效地执行。在该超步中，删除会首先被执行，先删除边，后删除顶点，因为顶点的删除通常也意味着删除其所有的出射边。然后执行添加操作，先增加顶点，后增加边，并且所有的拓扑改变都会在 Compute()函数调用前完成。这种局部有序保证了大多数冲突的结果的确定性。

剩余的冲突就需要通过用户自定义的 handlers 来解决。如果在一个超步中有多个请求需要创建一个相同的顶点，在默认情况下系统会随便挑选一个请求，但有特殊需求的用户可以定义一个更好的冲突解决策略，用户可以在 Vertex 类中通过定义一个适当的 handler 函数来解决冲突。同一种 handler 机制将被用于解决由于多个顶点删除请求或多个边增加请求或删除请求而造成的冲突。Pregel 委托 handler 来解决这种类型的冲突，从而使得 Compute()函数变得简单，而这样同时也会限制 handler 和 Compute()的交互，但这在应用中还没有遇到什么问题。

Pregel 的协同机制比较懒，全局的拓扑改变在被 apply 之前不需要进行协调，即在变更请求的发出端不会进行任何的控制协调，只有在它被接收到然后 apply 时才进行控制，这样就简化了流程，同时能让发送更快。这种设计的选择是为了优化流式处理。直观来讲，就是

对顶点 V 的修改引发的冲突由 V 自己来处理。

Pregel 同样也支持纯 local 的拓扑改变，例如，一个顶点添加或删除其自身的出射边或删除其自己。Local 的拓扑改变不会引发冲突，并且顶点或边的本地增减能够立即生效，很大程度上简化了分布式编程。

9.4.5 Input and Output

Pregel 可以采用多种文件格式进行图的保存，比如可以用 text 文件、关系数据库或者 Bigtable 中的行。为了避免规定死一种特定文件格式，Pregel 将“从输入中解析出图结构”这个任务从图的计算过程中进行了分离。类似地，结果可以以任何一种格式输出，并根据应用程序选择最适合的存储方式。Pregel library 本身提供了很多常用文件格式的 readers 和 writers，但是，用户可以通过继承 Reader 和 Writer 类来定义他们自己的读写方式。

9.5 Pregel 的基本体系结构

Pregel 是为 Google 的集群架构而设计的。每一个集群都包含了上千台机器，这些机器都分列在许多机架上，机架之间有着非常高的内部通信带宽。集群之间是内部互联的，但地理上是分布在不同地方的。

应用程序通常通过一个集群管理系统来执行，该管理系统会通过调度作业来优化集群资源的使用率，有时候会杀掉一些任务或将任务迁移到其他机器上去。该系统中提供了一个名字服务系统，所以，各任务间可以通过与物理地址无关的逻辑名称来各自标识自己。持久化的数据被存储在 GFS 或 Bigtable 中，而临时文件比如缓存的消息则存储在本地磁盘中。

9.5.1 Pregel 的执行过程

Pregel library 将一张图划分成许多的 partitions（如图 9-8 所示），每一个 partition 包含了一些顶点和以这些顶点为起点的边。将一个顶点分配到某个 partition 上去，取决于该顶点的 ID，这意味着，即使在别的机器上，也是可以通过顶点的 ID 来知道该顶点是属于哪个 partition，即使该顶点已经不存在了。默认的 partition 函数为 hash(ID) mod N，N 为所有 partition 总数，但是，用户可以替换掉它。

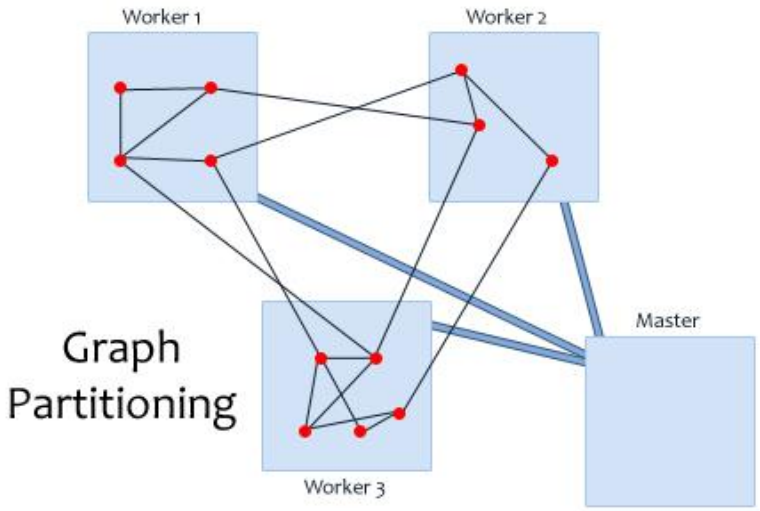

图 9-8 图的划分图

将一个顶点分配给哪个 worker 机器，是整个 Pregel 中对分布式不透明的主要地方。有些应用程序使用默认的分配策略就可以工作得很好，但是，有些应用可以通过定义一些可以更好地利用图本身的 locality 的分配函数而从中获益。比如，一种典型的可以用于 Web graph 的启发式方法是，将来自同一个站点的网页数据分配到同一台机器上进行计算。

在不考虑出错的情况下，一个 Pregel 程序的执行过程（如图 9-9 和图 9-10 所示）分为如下几个步骤：

1. 用户程序的多个 copy 开始在集群中的机器上执行。其中，有一个 copy 将会作为 master，其他的作为 worker，master 不会被分配图的任何一部分，而只是负责协调 worker 间的工作。worker 利用集群管理系统中提供的名字服务，来定位 master 位置，并发送注册信息给 master。

2. Master 决定对这个图需要多少个 partition，并分配一个或多个 partitions 到 worker 所在的机器上，如图 9-8 所示。这个数字也可能由用户进行控制。一个 worker 上有多个 partition 的情况下，可以提高 partitions 间的并行度，实现更好的负载平衡，通常都可以提高性能。每一个 worker 负责维护在其之上的图的那一部分的状态(顶点及边的增删)，对该部分中的顶点执行 Compute()函数，并管理发送出去的以及接收到的消息。每一个 worker 都知道该图的计算在所有 worker 中的分配情况。

3. Master 进程为每个 worker 分配用户输入中的一部分，这些输入被看作是一系列记录的集合，每一条记录都包含任意数目的顶点和边。对输入的划分和对整个图的划分是正交的，通常都是基于文件边界进行划分。如果一个 worker 加载的顶点刚好是这个 worker 所分配到

的那一部分，那么相应的数据结构就会被立即更新。否则，该 worker 就需要将它发送到它所应属于的那个 worker 上。当所有的输入都被 load 完成后，所有的顶点将被标记为 active 状态。

4. Master 给每个 worker 发指令，让其运行一个超步，worker 轮询在其之上的顶点，会为每个 partition 启动一个线程。调用每个 active 顶点的 Compute()函数，传递给它从上一次超步发送来的消息。消息是被异步发送的，这是为了使得计算和通信可以并行，以及进行 batching，但是，消息的发送会在本超步结束前完成。当一个 worker 完成了其所有的工作后，会通知 master，并告知当前该 worker 上在下一个超步中将还有多少 active 节点。不断重复该步骤，只要有顶点还处在 active 状态，或者还有消息在传输。

5. 计算结束后，master 会给所有的 worker 发指令，让它保存它那一部分的计算结果。

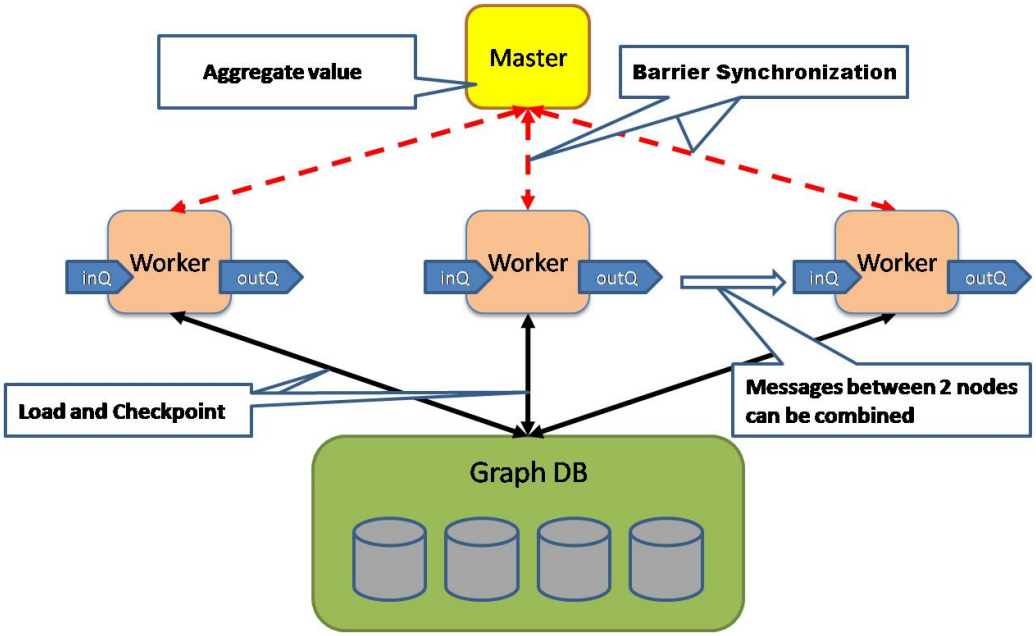

图 9-9 Pregel 的执行过程图

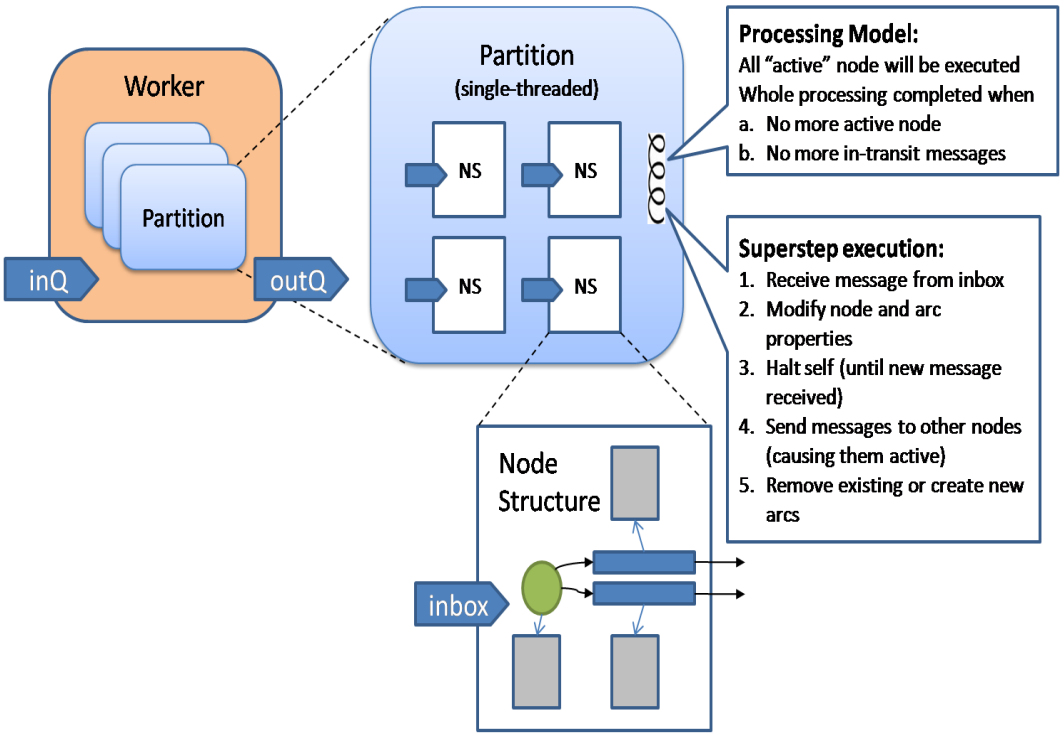

图 9-10 Worker 结构图

9.5.2 容错性

容错是通过检查点来实现的。在每个超步的开始阶段，master 命令 worker 让它保存它上面的 partitions 的状态到持久存储设备，包括顶点值、边值以及接收到的消息。Master 自己也会保存 aggregator 的值。

worker 的失效是通过 master 发给它的周期性的 ping 消息来检测的。如果一个 worker 在特定的时间间隔内没有收到 ping 消息，该 worker 进程会终止。如果 master 在一定时间内没有收到 worker 的反馈，就会将该 worker 进程标记为失败。

当一个或多个 worker 发生故障，被分配到这些 worker 的 partitions 的当前状态信息就丢失了。Master 重新分配图的 partition 到当前可用的 worker 集合上，所有的 partition 会从最近的某超步 S 开始时写出的检查点中，重新加载状态信息。该超步可能比在失败的 worker 上最后运行的超步 S'要早好几个阶段，此时，失去的几个超步将需要被重新执行。Pregel 对检查点频率的选择，是基于某个故障模型的平均时间的，从而平衡检查点的开销和恢复执行的开销。

为了改进恢复执行的开销和延迟， Confined recovery 已经在开发中。它们会首先通过检查点进行恢复，然后，系统会通过回放来自正常的 partitions 的记入日志的消息以及恢复

过来的 partitions 重新生成的消息，更新状态到 S’阶段。这种方式通过只对丢失的 partitions 进行重新计算，节省了在恢复时消耗的计算资源，同时，由于每个 worker 只需要恢复很少的 partitions，减少了恢复时的延迟。对发送出去的消息进行保存，会产生一定的开销，但是，通常机器上的磁盘带宽不会让这种 IO 操作成为瓶颈。

Confined recovery 要求用户算法是确定性的，以避免原始执行过程中所保存下的消息与恢复时产生的新消息并存情况下带来的不一致。随机化算法可以通过基于超步和 partition 产生一个伪随机数生成器来使之确定化。非确定性算法需要关闭 Confined recovery 而使用老的恢复机制。

9.5.3 Worker

一个 worker 机器会在内存中维护分配到其之上的 graph partition 的状态。从概念上来讲，可以简单地看作是一个从顶点 ID 到顶点状态的 Map，其中，顶点状态包括如下信息：该顶点的当前值，一个以该顶点为起点的出射边(包括目标顶点 ID，边本身的值)列表，一个保存了接收到的消息的队列，以及一个记录当前是否 active 的标志位。该 worker 在每个超步中，会循环遍历所有顶点，并调用每个顶点的 Compute()函数，传给该函数顶点的当前值，一个接收到的消息的迭代器和一个出射边的迭代器。这里没有对入射边的访问，原因是每一条入射边其实都是其源顶点的所有出射边的一部分，通常在另外的机器上。

出于性能的考虑，标志顶点是否为 active 的标志位，是和输入消息队列分开保存的。另外，只保存了一份顶点值和边值，但有两份顶点 active flag 和输入消息队列存在，一份是用于当前超步，另一个用于下一个超步。当一个 worker 在进行超步 S 的顶点处理时，同时还会有另外一个线程负责接收来自同一个超步的其他 worker 的消息。由于顶点当前需要的是 S-1 超步的消息，那么对超步 S 和超步 S+1 的消息就必须分开保存。类似地，顶点 V 接收到了消息，表示 V 将会在下一个超步中处于 active，而不是当前这一次。

当 Compute()请求发送一个消息到其他顶点时，worker 首先确认目标顶点是属于远程的 worker 机器，还是当前 worker。如果是在远程的 worker 机器上，那么消息就会被缓存，当缓存大小达到一个阈值，最大的那些缓存数据将会被异步地 flush 出去，作为单独的一个网络消息传输到目标 worker。如果是在当前 worker，那么就可以做相应的优化：消息就会直接被放到目标顶点的输入消息队列中。

如果用户提供了 Combiner，那么在消息被加入到输出队列或者到达输入队列时，会执

行 combiner 函数。后一种情况并不会节省网络开销，但是会节省用于消息存储的空间。

9.5.4 Master

Master 主要负责 worker 之间的工作协调，每一个 worker 在其注册到 master 的时候会被分配一个唯一的 ID。Master 内部维护着一个当前活动的 worker 列表，该列表中就包括每个 worker 的 ID 和地址信息，以及哪些 worker 被分配到了整个图的哪一部分。Master 中保存这些信息的数据结构大小，与 partitions 的个数相关，与图中的顶点和边的数目无关。因此，虽然只有一台 master，也足够用来协调对一个非常大的图的计算工作。

绝大部分的 master 的工作，包括输入 、输出、计算、保存以及从 checkpoint 中恢复，都将会在一个叫做 barriers 的地方终止。Master 在每一次操作时都会发送相同的指令到所有的活着的 worker，然后等待来自每个 worker 的响应。如果任何一个 worker 失败了，master 便进入恢复模式。如果 barrier 同步成功，master 便会进入下一个处理阶段，例如 master 增加超步的 index，并进入下一个超步的执行。

Master 同时还保存着整个计算过程以及整个 graph 的状态的统计数据，如图的总大小，关于出度分布的柱状图，处于 active 状态的顶点个数，在当前超步的时间信息和消息流量，以及所有用户自定义 aggregators 的值等。为方便用户监控，Master 在内部运行了一个 HTTP 服务器来显示这些信息。

9.5.5 Aggregators

每个 Aggregator 会通过对一组 value 值集合应用 aggregation 函数计算出一个全局值。每一个 worker 都保存了一个 aggregators 的实例集，由 type name 和实例名称来标识。当一个 worker 对 graph 的某一个 partition 执行一个超步时，worker 会 combine 所有的提供给本地的那个 aggregator 实例的值到一个 local value：即利用一个 aggregator 对当前 partition 中包含的所有顶点值进行局部归约。在超步结束时，所有 workers 会将所有包含局部归约值的 aggregators 的值进行最后的汇总，并汇报给 master。这个过程是由所有 worker 构造出一棵归约树而不是顺序地通过流水线的方式来归约，这样做的原因是为了并行化归约时 CPU 的使用。在下一个超步开始时，master 就会将 aggregators 的全局值发送给每一个 worker。

9.6 Pregel 的应用实例

9.6.1 最短路径

最短路径问题是图论中最有名的问题之一了，同时具有广泛的应用。该问题有几个形式：（1）单源最短路径，是指要找出从某个源顶点到其他所有顶点的最短路径；（2）s-t 最短路径，是指要找出给定源顶点 s 和目标顶点 t 间的最短路径，这个问题具有广泛的实验应用，比如寻找驾驶路线，并引起了广泛关注，同时它也是相对简单的；（3）全局最短路径，对于大规模的图对象来说，通常都不太实际，因为它的空间复杂度是 O(V*V)的。为了简化起见，我们这里以非常适于用 Pregel 解决的单源最短路径为例，实现代码如下：

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
      int mindist = IsSource(vertex_id()) ? 0 : INF;
      for (; !msgs->Done(); msgs->Next())
        mindist = min(mindist, msgs->Value());
      if (mindist < GetValue()) {
        *MutableValue() = mindist;
        OutEdgeIterator iter = GetOutEdgeIterator();
        for (; !iter.Done(); iter.Next())
            SendMessageTo(iter.Target(),
                          mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

在该算法中，我们假设与顶点关联的那个值被初始化为 INF。在每个超步中，每个顶点会首先接收到来自邻居传送过来的消息，该消息包含更新过的、从源顶点到该顶点的潜在的最短距离。如果这些更新里的最小值小于该顶点当前关联值，那么顶点就会更新这个值，并

发送消息给它的邻居。在第一个超步中，只有源顶点会更新它的关联值，然后发送消息给它的直接邻居。然后这些邻居会更新它们的关联值，然后继续发送消息给它们的邻居，如此循环往复。当没有更新再发生的时候，算法就结束，之后所有顶点的关联值就是从源顶点到它的最短距离，若值为 INF 表示该顶点不可达。如果所有的边权重都是非负的，就可以保证该过程肯定会结束。

该算法中的消息保存都是潜在的最小距离。由于接收顶点实际上只关注最小值，因此该算法是可以通过 combiner 进行优化的，combiner 实现代码如下所示，它可以大大减少 worker 间的消息量，以及在执行下一个超步前所需要缓存的数据量。

```
class MinIntCombiner : public Combiner<int> {
    virtual void Combine(MessageIterator* msgs) {
        int mindist = INF;
        for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
        Output("combined_source", mindist);
    }
};
```

与其他类似的串行算法（比如 Dijkstra 或者 Bellman-Ford）相比，该算法需要更多的比较次数，但是，它可以用来解决对于单机版实现很难解决的那个规模上的最短路径问题。还有一些更高级的并行算法，比如 Thorup 和Δ -stepping 算法，这些高级算法也可以在 Pregel 系统中实现。但是，上述代码的实现，由于其比较简单同时性能也还可以接受，对于那些普通用户来说也还是很具有吸引力的。

9.6.2 二分匹配

二分匹配算法的输入由两个不同的顶点集合组成，所有边的两个顶点分别位于两个集合中，输出是边的一个子集，它们之间没有公共顶点。极大匹配(Maximal Matching)是指在当前已完成的匹配下，无法再通过增加未完成匹配的边的方式来增加匹配的边数。Pregel 实现了一个随机化的极大匹配算法以及一个最大权匹配算法；下面是随机化的极大匹配算法。

```
Class BipartiteMatchingVertex
    : public Vertex<tuple<position, int>, void, boolean> {
  public:
    virtual void Compute(MessageIterator* msgs) {
      switch (superstep() % 4) {
        case 0: if (GetValue().first == 'L') {
            SendMessageToAllNeighbors(1);
            VoteToHalt();
          }
        case 1: if (GetValue().first == 'R') {
          Rand myRand = new Rand(Time());
          for ( ; !msgs->Done(); msgs->Next()) {
            if (myRand.nextBoolean()) {
              SendMessageTo(msgs->Source, 1);
              break;
            }
          }
          VoteToHalt(); }
        case 2:
            if (GetValue().first == 'L') {
              Rand myRand = new Rand(Time());
              for ( ; !msgs->Done(); msgs->Next) {
                if (myRand.nextBoolean()) {
                  *MutableValue().second = msgs->Source());
                  SendMessageTo(msgs->Source(), 1);
                  break;
                }
              }
              VoteToHalt(); }
```

```
            case 3:
                if (GetValue().first == 'R') {
                    msgs->Next();
                    *MutableValue().second = msgs->Source();
                }
                VoteToHalt();
        }}};
```

在该算法的 Pregel 实现中，顶点的关联值是由两个值组成的元组(tuple)：一个是用于标识该顶点所处集合(L or R)的 flag，一个是跟它所匹配的顶点名称。边的关联值类型为 void，消息的类型为 boolean。该算法是由四个阶段组成的多个循环组成（如图 9-11 所示），用来标识当前所处阶段的 index 可以通过用当前超步的 index mod 4 得到。

在循环的阶段 0，左边集合中那些还未被匹配的顶点会发送消息给它的每个邻居请求匹配，然后会无条件地 VoteToHalt。如果它没有发送消息(可能是因为它已经找到了匹配，或者没有出射边)，或者是所有的消息接收者都已经被匹配，该顶点就不会再变为 active 状态。

在循环的阶段 1，右边集合中那些还未被匹配的顶点随机选择它接收到的消息中的其中一个，并发送消息表示接受该请求，然后给其他请求者发送拒绝消息。然后，它也无条件地 VoteToHalt。

在循环的阶段 2，左边集合中那些还未被匹配的顶点选择它所收到右边集合发送过来的接受请求中的其中一个，并发送一个确认消息。左边集合中那些已经匹配好的顶点永远都不会执行这个阶段，因为它们不会在阶段 0 发送任何消息。

在循环的阶段 3，右边集合中还未被匹配的顶点最多会收到一个确认消息。它会通知匹配顶点，然后无条件地 VoteToHalt，它的工作已经完成。

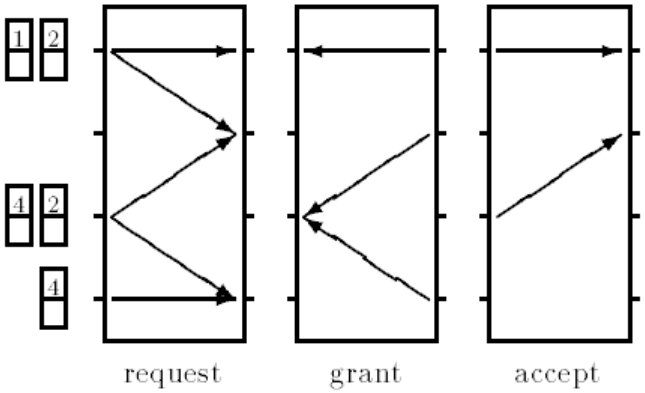

图 9-11 循环的四个阶段

9.7 改进的图计算模型

9.7.1 Pregel 的不足之处

作为第一个通用的大规模图处理系统，Pregel 已经为分布式图处理迈进了不小的一步，这点不容置疑，但是，Pregel 在一些地方也不尽如人意，具体如下：

1.在图的划分上，采用的是简单的 hash 方式，这样固然能够满足负载均衡，但是 hash 方式并不能根据图的连通特性进行划分，导致超步之间的消息传递开销可能会是影响性能的最大隐患。

2.简单的 checkpoint 机制只能向后式地将状态恢复到当前 S 超步的几个超步之前，要到达 S 还需要重复计算，这其实也浪费了很多时间，因此，如何设计 checkpoint，使得只需重复计算故障 worker 的 partition 的计算，从而节省计算，甚至可以通过 checkpoint 直接到达故障发生前一超步 S，也是一个很需要研究的地方。

3.BSP 模型本身有其局限性，整体同步并行对于计算快的 worker 长期等待的问题，仍然无法解决。

4.由于 Pregel 目前的计算状态都是常驻内存的，对于规模继续增大的图处理可能会导致内存不足，如何解决尚待研究。

9.7.2 PowerGraph

PowerGraph 将基于 vertex 的图计算抽象成一个通用的计算模型：GAS 模型（代码实现如下），分为三个阶段：Gather、Apply 和 Scatter。

1. Gather 阶段：用户自定义一个 sum 操作，用于各个顶点，将顶点的相邻顶点和对应边收集起来；

2. Apply 阶段：各个顶点利用上一阶段的 sum 值进行计算，并更新原始值；

3. Scatter 阶段：利用第二阶段的计算结果，更新顶点相连的边的值。

```
// gather_nbrs: IN_NBRS
gather(Du, D(u,v), Dv):
    return Dv.rank / #outNbrs(v)
```

```
    sum(a, b): return a + b
    apply(Du, acc):
        rnew = 0.15 + 0.85 * acc
        Du.delta = (rnew - Du.rank)/
                    #outNbrs(u)
        Du.rank = rnew
    // scatter_nbrs: OUT_NBRS
    scatter(Du,D(u,v),Dv):
        if(|Du.delta|>) Activate(v)
  return delta
```

由于顶点计算会频繁调用 Gather 阶段操作，而大多数相邻的顶点的值其实并不会变化，为了减少计算量，PowerGraph 提供了一种 Cache 机制，上面显示了 PowerGraph 机制下 Page Rank 计算的过程伪代码。

PowerGraph 提出了一种均衡图划分方案，减少计算中通信量的同时保证负载均衡。与 Pregel 和 GraphLab 均采用的 hash 随机分配方案不同，它提出了一种均衡 p-路顶点切割（vertex-cut）分区方案。根据图的整体分布概率密度函数计算顶点切割的期望值：

$$\mathbb{E}\left[\left(1-\frac{1}{p}\right)^{\mathbf{D}[v]}\right]=\frac{1}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}\left(1-\frac{1}{p}\right)^{d}d^{-\alpha}.$$

根据该期望值指导对顶点进行切割，并修改了传统的通信过程，具体如图 9-12 所示。

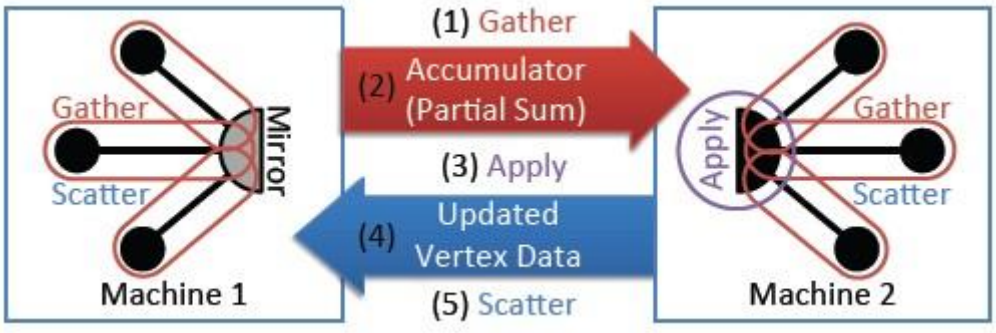

图 9-12 通信过程图

实验时，PowerGraph 按同步方式不同分别实现了三种版本（全局同步、全局异步和可串行化异步），具体如下：

- **全局同步**：与 Pregel 类似，超步之间设置全局同步点，用以同步对所有边以及顶点的

修改；

- **全局异步**：类似 GraphLab，所有 Apply 阶段和 Scatter 阶段对边或顶点的修改立即更新到图中；
- **可串行化异步**：全局异步会使得算法设计和调试变得很复杂，对某些算法效率可能比全局同步还差，因此有全局异步加可串行化结合的方式。

在差错控制上，依靠 checkpoint 的实现，采用 GraphLab 中使用的 Chandy-Lamport 快照算法。

通过将图计算模型进行抽象，设计实现均衡的图划分方案，对比三种不同方式下的系统实现，并实现了差错控制。

本章小结

本章首先简单介绍了图计算中的问题，介绍了处理图计算的两种软件，接着介绍了 Pregel 和 MapReduce 两种框架在 PageRank 算法中的实现，主要为了说明 Pregel 处理图计算问题的优势。然后主要重点介绍了 Pregel 图计算模型，它的 C++ API、执行过程和算法实现。最后介绍了一种对 Pregel 改进的图计算框架 PowerGraph。

参考文献

[1] Pregel—— 大规模图处理系统 http://www.cnblogs.com/panfeng412/archive/2011/10/28/2227195.html

[2]PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs(OSDI'12) http://wuyanzan60688.blog.163.com/blog/static/127776163201312863021180/

[3] PageRank 算法在 Pregel 和 MapReduce 两种计算模型中的思路. http://wuyanzan60688.blog.163.com/blog/static/127776163201211043525435/

[4] 被冷落的大数据热点：图谱分析. http://www.ctocio.com/ccnews/12340.html

[5] 论文 Pregel: A System for Large-Scale Graph Processing.

第 10 章 NoSQL 数据库

NoSQL 数据库，指的是非关系型的数据库。随着互联网 web2.0 网站的兴起，传统的关系数据库在应付 web2.0 网站，特别是超大规模和高并发的 SNS 类型的 web2.0 纯动态网站方面，已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。

本章介绍 NoSQL 数据库相关知识，内容要点如下：

- NoSQL 简介
- NoSQL 现状
- 为什么要使用 NoSQL 数据库
- NoSQL 数据库的特点
- NoSQL 的五大挑战
- 对 NoSQL 的质疑
- NoSQL 的三大基石
- NoSQL 数据库与关系数据库的比较
- 典型的 NoSQL 数据库分类
- NoSQL 数据库开源软件

10.1 NoSQL 简介

NoSQL，意即反 SQL 运动，是一项全新的数据库革命性运动，早期就有人提出，发展至 2009 年趋势越发高涨。NoSQL 的拥护者们提倡运用非关系型的数据存储，相对于目前铺天盖地的关系型数据库运用，这一概念无疑是一种全新的思维的注入。反 SQL 运动的主要倡导者都是 Web 和 Java 开发者，他们中许多人都在创业的初期历经了资金短缺并因此与 Oracle 说再见，然后效仿 Google 和 Amazon 的道路建设起自己的数据存储解决方案，并随

后将自己的成果开源发布。现在，他们的开源数据商店管理着成百 TB 甚至 PB 的数据，由于 Web 2.0 和云计算的兴起，无论从技术上还是从经济上他们都无需再返回从前，甚至连想也不用想。

“Web 2.0 的企业应该抓住机会，他们需要可扩展性”，总部设在伦敦的 NoSQL 会议组织者 Johan Oskarsson 说。Oskarsson 任职于著名的音乐网站 Last.fm，其他的大多数与会者也都是网络开发者。

Oskarsson 说，许多人甚至抛弃了 MySQL 开源数据库这个长期以来 Web 2.0 的宠儿，而改由 NoSQL 的方案来替代，因为优势实在是引人注目。过度的商业化使得 MySQL 失去原来的优势。

例如，Facebook 建立了自己的 Cassandra 数据商店，并且在其网站上重点推出一项新的搜索功能，没有使用到现有的 MySQL 数据库。据 Facebook 的工程师 Avinash Lakshma 介绍，Cassandra 仅用 0.12 毫秒就可以写入 50GB 的数据，比 MySQL 快了超过 2500 倍。Google 也开始公测他们的云数据库 Fusion Tables，这是一个和传统数据库完全不同的数据库，主要优势在于能够简单地解决关系型数据库中管理不同类型数据的麻烦，以及排序整合的常见操作的性能问题等。

10.2 NoSQL 现状

现今的计算机体系结构在数据存储方面要求具备庞大的水平扩展性（即能够连接多个软硬件的特性，这样可以将多个服务器从逻辑上看成一个实体），而 NoSQL 致力于改变这一现状。目前 Google 的 BigTable 和 Amazon 的 Dynamo 使用的就是 NoSQL 型数据库。

NoSQL 项目的名字上看不出什么相同之处，但是，它们通常在某些方面相同：它们可以处理超大量的数据。

这场革命目前仍然需要等待。的确，NoSQL 对大型企业来说还不是主流，但是，一两年之后很可能就会变个样子。在 NoSQL 运动的最新一次聚会中，来自世界各地的 150 人挤满了 CBS Interactive 的一间会议室，分享他们如何推翻缓慢而昂贵的关系数据库的“暴政”，怎样使用更有效和更便宜的方法来管理数据。

“关系型数据库给你强加了太多东西。它们要你强行修改对象数据，以满足 RDBMS（relational database management system，关系型数据库管理系统）的需要”，在 NoSQL 拥护者们看来，基于 NoSQL 的替代方案“只是给你所需要的”。

虽然有些人认为这是摆脱 MySQL 和 PostgreSQL 等传统的开源关系数据库的机会，实际上事情并不是这么简单，从这些有趣的变化中我们得出一些启示：

1）关系数据库并不适合所有的数据模型；

2）关系数据库扩展难度大，特别是当你一开始就设计为单机配置，未进行分布式设计时；

3）标准化通常会伤害到性能；

4）在许多应用中，主键就是你的一切。

新的 NoSQL 数据存储完全改变了传统的观念，但总的来说，它们借鉴了一套类似的高级特征，但它们并非能够满足一切。下面给出一个列表，让我们来看看 NoSQL 正试图实现什么：

1）反标准化，通常是无模式的，文档型存储；

2）以 key/value 为基础，支持通过 key 进行查找；

3）水平扩展；

4）内置复制；

5）HTTP/REST 或很容易编程的 API；

6）支持 MapReduce 的风格的编程；

7）最终一致性。

如果还要列的话，可能还可以列出一打来。但前面两个是对传统数据库最大的叛离，当然你也可以坚持使用 MySQL，并将其去关系化，这也是 FriendFeed 要做的事情，FriendFeed 使用 MySQL 作为后端，实现分布式 key/value 存储。对这些分布式无模式的数据存储，开始有一个新名称来称呼，那就是 NoSQL。

10.3 为什么要使用 NoSQL 数据库?

随着互联网 web2.0 网站的兴起，非关系型的数据库现在成了一个极其热门的新领域，非关系数据库产品的发展非常迅速。而传统的关系数据库在应付 web2.0 网站，特别是超大规模和高并发的 SNS 类型的 web2.0 纯动态网站方面，已经显得力不从心，暴露了很多难以克服的问题，主要包括以下几个方面：

- **对数据库高并发读写的性能需求：**web2.0 网站要根据用户个性化信息来实时生成动态页面和提供动态信息，所以，基本上无法使用动态页面静态化技术，因此数

据库并发负载非常高，往往要达到每秒上万次读写请求。关系数据库应付上万次 SQL 查询还勉强顶得住，但是应付上万次 SQL 写数据请求，硬盘 IO 就已经无法承受了。其实对于普通的 BBS 网站，往往也存在对高并发写请求的需求。

- **对海量数据的高效率存储和访问的需求：**对于大型的 SNS 网站，每天用户产生海量的用户动态，以国外的 Friendfeed 为例，一个月就达到了 2.5 亿条用户动态，对于关系数据库来说，在一张 2.5 亿条记录的表里面进行 SQL 查询，效率是极其低下甚至是不可忍受的。再例如大型 web 网站的用户登录系统，例如腾讯和盛大，动辄数以亿计的帐号，关系数据库也很难应付。
- **对数据库的高可扩展性和高可用性的需求：**在基于 web 的架构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，你的数据库却没有办法像网页服务器和应用服务器那样简单地通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供 24 小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移，为什么数据库不能通过不断地添加服务器节点来实现水平扩展呢？

在上面提到的“三高”需求面前，关系数据库遇到了难以克服的障碍，而对于 web2.0 网站来说，关系数据库的很多主要特性却往往无用武之地，主要表现在以下几个方面：

- **数据库事务一致性需求：**很多 web 实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求也不高。因此，数据库事务管理成了数据库高负载下一个沉重的负担。
- **数据库的写实时性和读实时性需求：**对于关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多 web 应用来说，并不要求这么高的实时性。
- **对复杂的 SQL 查询，特别是多表关联查询的需求：**任何大数据量的 web 系统，都非常忌讳多个大表的关联查询以及复杂数据分析类型的复杂 SQL 报表查询，特别是 SNS 类型的网站，往往从需求以及产品设计角度就避免了这种情况的产生。一般而言，这类 web 系统更多的只是单表的主键查询以及单表的简单条件分页查询，SQL 的功能被极大地弱化了。

因此，关系数据库在这些越来越多的应用场景下就显得不那么合适了，为了解决这类问题的非关系数据库由此应运而生。

NoSQL 是非关系型数据存储的广义定义，它打破了长久以来关系型数据库与 ACID 理

论大一统的局面。NoSQL 数据存储不需要固定的表结构，通常也不存在连接操作。在大数据存取上具备关系型数据库无法比拟的性能优势。该术语在 2009 年初得到了广泛的认同。

当今的应用体系结构需要数据存储在横向伸缩性上能够满足需求。而 NoSQL 存储就是为了实现这个需求。Google 的 BigTable 与 Amazon 的 Dynamo 是非常成功的商业 NoSQL 实现。一些开源的 NoSQL 体系，如 Facebook 的 Cassandra，Apache 的 HBase，也得到了广泛的认同。从这些 NoSQL 项目的名字上看不出什么相同之处，比如 Hadoop、Voldemort、Dynomite 等，当然，还存在其它很多 NoSQL 项目。

10.4 NoSQL 数据库的特点

NoSQL 数据库的主要特点包括以下几个方面：

- **灵活的可扩展性**

多年以来，数据库管理员们都是通过“纵向扩展”的方式（当数据库的负载增加的时候，购买更大型的服务器来承载增加的负载）来进行扩展的，而不是通过“横向扩展”的方式（当数据库负载增加的时候，在多台主机上分配增加的负载）来进行扩展。但是，随着交易率和可用性需求的增加，数据库也正在迁移到云端或虚拟化环境中，“横向扩展”在商用硬件方面的经济优势变得更加明显了，对各大企业来说，这种“诱惑”是无法抗拒的。在商业硬件集群上，要对 RDBMS 做“横向扩展”，并不是很容易，但是，各种新类型的 NoSQL 数据库主要是为了进行透明的扩展来利用新节点而设计的，而且，它们通常都是为了低成本的商用硬件而设计的。

- **大数据**

在过去的十年里，正如交易率发生了翻天覆地的增长一样，需要存储的数据量也发生了急剧的膨胀。O’Reilly 把这种现象称为“数据的工业革命”。为了满足数据量增长的需要，RDBMS 的容量也在日益增加，但是，对于一些企业来说，随着交易率的增加，单一数据库需要管理的数据约束的数量也变得越来越让人无法忍受了。现在，大量的“大数据”可以通过 NoSQL 系统（例如：Hadoop）来处理，它们能够处理的数据量远远超出了最大型的 RDBMS 所能处理的极限。

- **“永别了”？DBA 们！**

在过去的几年里，虽然一些 RDBMS 供应商们声称在可管理性方面做出了很多的改进，但是，高端的 RDBMS 系统维护起来仍然十分昂贵，而且还需要训练有素的 DBA 们的协助。

DBA 们需要亲自参与高端的 RDBMS 系统的设计、安装和调优。NoSQL 数据库从一开始就是为了降低管理方面的要求而设计的；从理论上来说，自动修复、数据分配和简单的数据模型，的确可以让管理和调优方面的要求降低很多。但是，DBA 的“死期将至”的谣言未免有些过于夸张了，毕竟总是需要有人对关键性的数据库的性能和可用性负责的。

- **经济**

NoSQL 数据库通常使用廉价的商用服务器集群来管理膨胀的数据和事务数量，而 RDBMS 通常需要依靠昂贵的专有服务器和存储系统来做到这一点。使用 NoSQL，每 GB 的成本或每秒处理的事务的成本，都比使用 RDBMS 的成本少很多倍，这可以让你花费更低的成本存储和处理更多的数据。

- **灵活的数据模型**

对于大型的生产性的 RDBMS 来说，变更管理是一件很令人头痛的事情。即使只对一个 RDBMS 的数据模型做了很小的改动，也必须要十分小心地管理，也许还需要停机或降低服务水平。NoSQL 数据库在数据模型约束方面是更加宽松的，甚至可以说并不存在数据模型约束。NoSQL 的键值数据库和文档数据库，可以让应用程序在一个数据元素里存储任何结构的数据。即使是规定更加严格的基于“大表”的 NoSQL 数据库（比如 Cassandra 和 HBase），通常也允许创建新列，这并不会造成什么麻烦。应用程序变更和数据库模式的变更，并不需要作为一个复杂的变更单元来管理。从理论上来说，这可以让应用程序迭代得更快，但是，很明显，如果应用程序无法维护数据的完整性，那么这会带来一些不良的副作用。

10.5 NoSQL 的五大挑战

NoSQL 的种种承诺引发了一场热潮，但是，在它们得到主流的企业的青睐以前，它们还有许多困难需要克服。下面是 NoSQL 需要面对的一些挑战：

- **成熟度**

RDBMS 系统已经发展很长时间了。NoSQL 的拥护者们认为，RDBMS 系统那超长的发展的年限，恰恰表示它们已经过时了；但是，对于大多数的 CIO 们来说，RDBMS 的成熟度更加令它们放心。大多数情况下，RDBMS 系统更加稳定，而且功能也更加丰富。相比之下，大多数的 NoSQL 数据库都是“前期制作”版本，许多关键性的功能还有待实现。对于大多数开发者来说，处于技术的最前沿的确是很令人兴奋的，但是，企业应该怀着极端谨慎的态度来处理此事。

- **技术支持**

企业都希望能得到这样的保证：如果一个关键性的系统出现问题了，他们可以获得及时有效的技术支持。所有的 RDBMS 供应商都在竭尽全力地提供高水平的企业技术支持。相反，大多数的 NoSQL 系统都是开源项目，虽然对于每个 NoSQL 数据库来说，通常也会有一个或多个公司对它们提供支持，但是，那些公司通常是小型的创业公司，在支持的范围、支持的资源或可信度方面，它们和 Oracle、Microsoft 或 IBM 是无法相提并论的。

- **分析和商业智能化**

NoSQL 数据库现在已经可以满足现代的 Web2.0 应用程序的高度的可扩展性的要求了，这直接导致的结果是，它们的大多数功能都是面向这些应用程序而设计的。但是，在一个应用程序中，具有商业价值的数据早就已经超出了一个标准的 Web 应用程序需要的“插入-读取-更新-删除”的范畴了。在公司的数据库中进行商业信息的挖掘，可以提高企业的效率和竞争力，而且，对于所有的中到大型的公司来说，商业智能化（BI）一直是一个至关重要的 IT 问题。NoSQL 数据库几乎没有提供什么专用的查询和分析工具，即使是一个简单的查询，也要求操作者具有很高超的编程技术，而且，常用的 BI 工具是无法连接到 NoSQL 的。像 Hive 或 Pig 那样的新出现的一些解决方案，在这个方面或许可以提供一些帮助，它们可以让访问 Hadoop 集群中的数据变得更加容易，最后也许还会支持其他的 NoSQL 数据库。Quest 软件已经开发了一个产品——Toad for Cloud Databases——它给各种 NoSQL 数据库提供了专用的查询功能。

- **管理**

NoSQL 的设计目标是提供一个“零管理”的解决方案，但是，就目前而言，还远远没有达到这个目标。安装 NoSQL 还是需要很多技巧的，同时，维护它也需要付出很多的努力。

- **专业知识**

毫不夸张地说，全世界有数百万的开发者，他们都对 RDBMS 的概念和编程方法很熟悉，在每个业务部门中都有这样的开发者。相反，几乎每一个 NoSQL 开发者都正处于学习状态中。虽然这种情况会随着时间的推移而改变，但是现在，找到一些有经验的 RDBMS 程序员或管理员要比找到一个 NoSQL 专家容易得多。

10.6 对 NoSQL 的质疑

不可否认，NoSQL 拥有众多的支持者，但是，这里也不妨让我们聆听一下对 NoSQL 质

疑的声音。有业界人士指出，NoSQL 这个项目的背景是站不住脚的。基于 SQL 的关系型数据库，确实在性能上存在一些瓶颈，但是，这大部分并不是这门 SQL 技术所造成的，而是因为在设计数据库的时候，表与表之间的关系、表的索引或者表空间的部署等等没有设计好而造成的。所以，关系型数据库性能不理想，并不能全部怪罪到这个技术本身上。通常情况下，对原有的数据库设计进行优化，往往可以在很大程度上提升数据库的性能。

有些业界人士仍然不是很看好 NoSQL 项目的前景，甚至有些人对其前途感到很悲观，认为 NoSQL 项目很难跟传统的关系型数据库相抗衡，甚至其想达到 MySQL 这个开源数据库的高度都很难。NoSQL 的质疑者主要从以下几个方面考虑问题：

- **NoSQL 很难实现数据的完整性**

很多关系型数据库中优秀的、实用的功能，在 NoSQL 数据库却无法实现。比如，在任何一个关系型的数据库中，都可以很容易地实现数据的完整性。如在 Oracle 数据库中，可以轻而易举地实现实体完整性(通过主键或者非空约束来实现)、参照完整性(通过主键、外键来实现)、用户定义完整性(通过约束或者触发器来实现)。

NoSQL 支持者也承认关系型数据库在数据完整性上的作用是不可替代的。但是他们却反驳说，企业可能用不到这么复杂的功能。对于这一点，很多人是不敢认同的。现在企业的任何一个应用，基本上都需要用到数据完整性。如现在大部分应用至少都需要有一个用户认证的过程。为此，在系统实现的过程中，需要在数据库中保存用户名。由于这个用户名涉及到用户的认证问题，为此用户名必须要唯一，此时就需要用到唯一性约束。在关系型数据库中，只需要在表格设计过程中，将用户名设置为唯一即可。而在 NoSQL 中，还需要通过代码来实现唯一性。本来很容易就可以实现，现在却要绕个弯去实现，这有点不可思议。由于在 NoSQL 项目中很难实现数据的完整性，而在企业应用中这个数据完整性又是少不了的。因此，我们有理由认为，NoSQL 项目很难在企业中普及开来。至少在短时间内，NoSQL 革命仍然需等待。

- **缺乏强有力的技术支持**

到目前为止，NoSQL 项目都是开源的。所以说，他们缺乏供应商技术人员提供的正式支持。在这一点上，NoSQL 项目与大多数的开源项目一样，不得不从社区中寻求支持。但是，NoSQL 项目比其他的开源项目要难得多。首先，NoSQL 项目是一个数据库系统的项目，或者说，是一些网络应用的最基层的设施。如果其出错的话，后果很严重。由于缺乏正式的官方支持，万一数据库运行出现了错误，后果是很严重的。而且到时候用户也是投诉无门的。所以，现在 NoSQL 项目基本上还是属于研究的阶段，如果要普及到大量企业中使用，被数

据库管理员所接受，至少其稳定性上要有所改善。或者说，当问题出现时，数据库管理员要能够及时修复运行故障。由于缺乏强有力的技术支持，数据库管理员担心故障出现时难以迅速解决，所以，很多管理员都拒绝使用 NoSQL 项目，即使其是开源免费的。如 NoSQL 项目的组织者 Oskarsson 也坦言，他们自己的公司现在使用的也不是 NoSQL 数据库，甚至在短期内也没有这个打算。他们现在使用的虽然是开源的数据库系统，但是，仍然是基于 SQL 的关系型数据库。

- **开源数据库从出现到被用户接受需要一个漫长的过程**

假设这个 NoSQL 技术能够被企业用户所接受，但是，从其出现到被用户最终接受需要一个漫长的过程。如 MySQL 这个开源的数据库系统，其从出现到流行也是花了好多年的时间。而且，MySQL 数据库是基于比较成熟的关系数据库模型的。其在开发设计的时候，已经有不少完善的产品可以参考。至少 SQL 语句的语法其可以直接拿来使用，而不用从零开始设计。而现在 NoSQL 是一个从零开始的产品，所有内容都需要重新设计。在没有供应商技术人员的支持下，这个过程可能是很漫长的。即使退一万步来说，最终其可以向 MySQL 数据库那样受中小企业的欢迎，但是，由于其自身技术的薄弱，在大型的数据库应用中就会显得心有余而力不足。

- **关系型数据库在设计时更能够体现实际**

其实，关系型数据库也是从非关系型数据库升级过来的。现在大部分数据库都是建立在关系型数据库模型之上的，这恰好说明了关系型数据库存在的价值。关系型数据库最大的价值就在于其设计方便。因为其数据库对象之间的关系模型(如第三范式等等)，对于数据库设计时很有帮助的，其在很大程度上体现了业务的实际情况。如在设计一个 ERP 系统时，主键与外键的关系可以反映出产品信息表与采购订单之间的关联。这种关系是那么地符合实际。而现在 NoSQL 项目想把这种关系剥离掉，那么在数据库设计的时候，必然会增加很多的麻烦，会增加数据库的难度。最重要的是，这些数据库对象之间的关系不仅仅是关系而已，其还是一种强有力的准则，对于所有的关系型数据库管理员都会产生约束。比如，Oracle 数据库的管理员经过简短的学习之后，也能够很快地掌握 SQLServer 数据库的技术，因为其内部的准则是共同的，数据库管理员只要学习其表现形式即可。这就好像学汽车，你只要拿出驾照，那么什么牌子的车都可以开。因为其数据库对象的关系、运行模式等等都是固定的。但是，NoSQL 项目由于缺乏这种关系，导致基于 NoSQL 技术的不同产品之间可能会存在很大的差异。这不仅在数据库设计的时候会增加不少的难度。而且在维护的时候，也需要花费更多的时间与精力。

总之，照目前的情况来看，有些业界人士对 NoSQL 项目的思路仍然是反对的。至少在近期很难有像样的 NoSQL 产品面世。NoSQL 项目的组织者 Oskarsson 也承认，NoSQL 项目这场数据库革命仍然需要等待。在短时间内，无法跟关系型数据库相互抗衡，也许永远没有这个机会。

10.7 NoSQL 的三大基石

NoSQL 的三大基石包括：CAP、BASE、最终一致性。

10.7.1 CAP

2000 年，Eric Brewer 教授指出了著名的 CAP 理论，后来 Seth Gilbert 和 Nancy lynch 两人证明了 CAP 理论的正确性。所谓的 CAP 指的是：

- **C（Consistency）：**一致性；
- **A:（Availability）：**可用性(指的是快速获取数据)；
- **P（Tolerance of network Partition）：**分区容忍性(分布式) 。

CAP 理论告诉我们，一个分布式系统不可能同时满足一致性、可用性和分区容错性这三个需求，最多只能同时满足其中两个。

正所谓“鱼和熊掌不可兼得也”。如果你关注的是一致性，那么你就需要处理因为系统不可用而导致的写操作失败的情况；而如果你关注的是可用性，那么你应该知道系统的读操作可能不能精确地读取到写操作写入的最新值。因此，系统的关注点不同，相应采用的策略也是不一样的，只有真正地理解了系统的需求，才有可能利用好 CAP 理论。

CAP 理论认为，在一个系统中，对于某个数据而言不存在一个算法同时满足 Consistency、Availability 和 Partition-tolerance。 注意，这里边最重要和最容易被人忽视的是限定词“对于某个数据而言不存在一个算法”。这就是说，在一个系统中，可以对某些数据做到 CP，对另一些数据做到 AP，就算是对同一个数据，调用者可以指定不同的算法，某些算法可以做到 CP，另一些算法可以做到 AP。

当处理 CAP 的问题时，可以有几个选择，最明显的是：

- **放弃 Partition Tolerance：**如果你想避免分区（partition）问题发生，你就必须要阻止其发生。一种做法是将所有的东西（与事务相关的）都放到一台机器上；另一种做法是，放在像 rack 这类的自动失败恢复单元上，当然，仍然无法 100%地保证不

发生失败，因为还是有可能部分失败，但是，你不太可能碰到由分区问题带来的负面效果。当然，这个选择会严重影响可扩展性。

- **放弃 Availability**：系统可以把这个数据只放在一个节点上，其他节点收到请求后向这个节点读或写数据，并返回结果。很显然，串行化是保证的。但是，如果报文可以任意丢失的话，接受请求的节点就可能永远不返回结果。
- **放弃 Consistency**：系统只要每次对写都返回成功，对读都返回固定的某个值就可以了。不同数据对于一致性的要求是不同的。举例来讲，用户评论对“不一致”是不敏感的，可以容忍相对较长时间的不一致，这种不一致并不会影响交易和用户体验。而产品价格数据则是非常敏感的，通常不能容忍超过 10 秒的价格不一致。对于大型网站而言，可用性与分区容忍性优先级要高于数据一致性，一般会尽量朝着 AP 的方向设计，然后，通过其它手段保证对于一致性的商务需求。架构设计师不要把精力浪费在如何设计能满足三者的完美分布式系统，而是应该进行取舍。
- **引入 BASE**：有一种架构的方法称作 BASE（Basically Available，Soft-state，Eventually consistent），支持最终一致性。BASE（注：化学中的含义是碱），如其名字所示，是 ACID（注：化学中的含义是酸）的反面。基于 BASE 实现一个满足最终一致性 (Eventually Consistency) 和 AP 的系统是可行的。现实中的一个例子是 Cassandra 系统。

10.7.2 BASE

说起来很有趣，BASE 的英文意义是碱，而 ACID 的英文含义是酸，看起来二者似乎是“水火不容”。BASE 的基本含义如下：

- **Basically Availble**：基本可用，支持分区失败；
- **Soft-state**：软状态/柔性事务，可以理解为"无连接"的，可以有一段时间不同步；而 "Hard state"是"面向连接"的；
- **Eventual Consistency**：最终一致性，也是 ACID 的最终目的，最终数据是一致的就可以了，而不是时时一致。

BASE 模型是反 ACID 模型的，完全不同于 ACID 模型，牺牲了高一致性，从而获得可用性或可靠性。BASE 思想主要强调基本的可用性，如果你需要高可用性，也就是纯粹的高性能，那么就要以一致性或容错性为牺牲，BASE 思想的方案在性能上还是有潜力可挖的。

10.7.3 最终一致性

对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性主要指的是多进程并发访问时更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为存在并发读写时才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。如果能容忍后续的部分或者全部访问不到，则是弱一致性。如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

最终一致性根据更新数据后各进程访问到数据的时间和方式的不同，又可以区分为：

- **因果一致性**：如果进程 A 通知进程 B 它已更新了一个数据项，那么进程 B 的后续访问将返回更新后的值。与进程 A 无因果关系的进程 C 的访问遵守一般的最终一致性规则。
- **“读己之所写（read-your-writes）”一致性**：当进程 A 自己更新一个数据项之后，它总是访问到更新过的值，绝不会看到旧值。这是因果一致性模型的一个特例。
- **会话（Session）一致性**：这是上一个模型的实用版本，它把访问存储系统的进程放到会话的上下文中。只要会话还存在，系统就保证“读己之所写”一致性。如果由于某些失败情形令会话终止，就要建立新的会话，而且系统的保证不会延续到新的会话。
- **单调（Monotonic）读一致性**：如果进程已经看到过数据对象的某个值，那么任何后续访问都不会返回在那个值之前的值。
- **单调写一致性**：系统保证来自同一个进程的写操作顺序执行。要是系统不能保证这种程度的一致性，就非常难以编程了。

上述最终一致性的不同方式可以进行组合，例如单调读一致性和读己之所写一致性就可以组合实现。并且从实践的角度来看，这两者的组合，读取自己更新的数据，和一旦读取到最新的版本不会再读取旧版本，对于此架构上的程序开发来说，会少很多额外的烦恼。

从服务端角度，如何尽快将更新后的数据分布到整个系统，降低达到最终一致性的时间窗口，是提高系统的可用度和用户体验非常重要的方面。对于分布式数据系统，定义如下参数：

- N — 数据复制的份数；
- W — 更新数据是需要保证写完成的节点数；
- R — 读取数据的时候需要读取的节点数；

如果 W+R>N，写的节点和读的节点重叠，则是强一致性。例如对于典型的一主一备同步复制的关系型数据库，N=2，W=2，R=1，则不管读的是主库还是备库的数据，都是一致的。

如果 W+R<=N，则是弱一致性。例如对于一主一备异步复制的关系型数据库，N=2，W=1，R=1，则如果读的是备库，就可能无法读取主库已经更新过的数据，所以是弱一致性。

对于分布式系统，为了保证高可用性，一般设置 N>=3。不同的 N、W、R 组合，是在可用性和一致性之间取一个平衡，以适应不同的应用场景。

如果 N=W，R=1，任何一个写节点失效，都会导致写失败，因此可用性会降低，但是由于数据分布的 N 个节点是同步写入的，因此可以保证强一致性。

如果 N=R，W=1，只需要一个节点写入成功即可，写性能和可用性都比较高。但是，读取其他节点的进程可能不能获取更新后的数据，因此是弱一致性。这种情况下，如果 W<(N+1)/2，并且写入的节点不重叠的话，则会存在写冲突。

10.8 NoSQL 数据库与关系数据库的比较

NoSQL 并没有一个准确的定义，但一般认为 NoSQL 数据库应当具有以下的特征：模式自由(schema-free)、支持简易备份(easy replication support)、简单的应用程序接口(simple API)、最终一致性(或者说支持 BASE 特性，不支持 ACID)、支持海量数据(Huge amount of data)。NoSQL 和关系型数据库的简单比较如表 10-1 所示。

表 10-1 NoSQL 和关系型数据库的简单比较

比较标准	RDBMS	NoSQL	备注

数据库原理	完全支持	部分支持	RDBMS 有数学模型支持、NoSQL 则没有
数据规模	大	超大	RDBMS 的性能会随着数据规模的增大而降低；NoSQL 可以通过添加更多设备以支持更大规模的数据
数据库模式	固定	灵活	使用 RDBMS 都需要定义数据库模式，NoSQL 则不用
查询效率	快	简单查询非常高效、较复杂的查询性能有所下降	RDBMS 可以通过索引，能快速地响应记录查询(point query)和范围查询(range query)；NoSQL 没有索引，虽然 NoSQL 可以使用 MapReduce 加速查询速度，仍然不如 RDBMS
一致性	强一致性	弱一致性	RDBMS 遵守 ACID 模型；NoSQL 遵守 BASE (Basically Available、soft state、Eventually consistent)模型
扩展性	一般	好	RDBMS 扩展困难；NoSQL 扩展简单
可用性	好	很好	随着数据规模的增大，RDBMS 为了保证严格的一致性，只能提供相对较弱的可用性；NoSQL 任何时候都能提供较高的可用性
标准化	是	否	RDBMS 已经标准化（SQL）；NoSQL 还没有行业标准
技术支持	高	低	RDBMS 经过几十年的发展，有很好的技术支持；NoSQL 在技术支持方面不如 RDBMS
可维护性	复杂	复杂	RDBMS 需要专门的数据库管理员(DBA)维护；NoSQL 数据库虽然没有 DBMS 复杂，也难以维护

10.9 典型的 NoSQL 数据库分类

典型的 NoSQL 数据库分类如表 10-2 所示。

表 10-2 典型的 NoSQL 数据库分类

NoSQL 数据库类型	代表性产品	性能	扩展性	灵活性	复杂性	优点	缺点
键/值数据库	Redis Riak	高	高	高	无	查询效率高	不能存储结构化信息
列式数据库	HBase Cassandra	高	高	一般	低	查询效率高	功能较少
文档数据库	CouchDB MongoDB	高	可变的	高	低	数据结构灵活	查询效率较低
图形数据库	Neo4J OrientDB	可变的	可变的	高	高	支持复杂的图算法	只支持一定的数据规模

10.10 NoSQL 数据库开源软件

10.10.1 Membase

Membase 是 NoSQL 家族的一个新的重量级的成员。Membase 是开源项目，源代码采用了 Apache2.0 的使用许可。该项目托管在 GitHub.Source tarballs 上，目前可以下载 beta 版本的 Linux 二进制包。该产品主要是由 North Scale 的 memcached 核心团队成员开发完成，其中还包括 Zynga 和 NHN 这两个主要贡献者的工程师，这两个组织都是很大的在线游戏和社区网络空间的供应商。

Membase 容易安装、操作，可以从单节点方便的扩展到集群，而且为 memcached（有线协议的兼容性）实现了即插即用功能，在应用方面为开发者和经营者提供了一个比较低的门槛。做为缓存解决方案，Memcached 已经在不同类型的领域（特别是大容量的 Web 应用）有了广泛的使用，其中 Memcached 的部分基础代码被直接应用到了 Membase 服务器的前

端。

通过兼容多种编程语言和框架，Membase 具备了很好的复用性。在安装和配置方面，Membase 提供了有效的图形化界面和编程接口，包括可配置的告警信息。

Membase 的目标是提供对外的线性扩展能力，包括为了增加集群容量，可以针对统一的节点进行复制。另外，对存储的数据进行再分配仍然是必要的。

这方面的一个有趣的特性是 NoSQL 解决方案所承诺的可预测的性能，类准确性的延迟和吞吐量。通过如下方式可以获得上面提到的特性：

- 自动将在线数据迁移到低延迟的存储介质的技术（内存，固态硬盘，磁盘）；
- 可选的写操作一一异步，同步（基于复制，持久化）；
- 反向通道再平衡（未来考虑支持）；
- 多线程低锁争用；
- 尽可能使用异步处理；
- 自动实现重复数据删除；
- 动态再平衡现有集群；
- 通过把数据复制到多个集群单元和支持快速失败转移来提供系统的高可用性。

10.10.2 MongoDB

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，是类似 json 的 bjson 格式，因此可以存储比较复杂的数据类型。Mongo 最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。它的特点是高性能、易部署、易使用，存储数据非常方便。

MongoDB 主要功能特性包括以下几个方面：

- 面向集合存储，易存储对象类型的数据：“面向集合”（Collenction-Orented），意思是数据被分组存储在数据集中，被称为一个集合（Collenction)。每个 集合在数据库中都有一个唯一的标识名，并且可以包含无限数目的文档。集合的概念类似关系型数据库（RDBMS）里的表（table），不同的是它不需要定 义任何模式（schema)。
- 模式自由：模式自由（schema-free)，意味着对于存储在 mongodb 数据库中的文件，我们不需要知道它的任何结构定义。如果需要的话，你完全可以把不同结构的文件

存储在同一个数据库里。

- 支持动态查询；
- 支持完全索引，包含内部对象；
- 支持查询；
- 支持复制和故障恢复；
- 使用高效的二进制数据存储，包括大型对象（如视频等）；
- 自动处理碎片，以支持云计算层次的扩展性；
- 支持 RUBY，PYTHON，JAVA，C++，PHP 等多种语言；
- 文件存储格式为 BSON（一种 JSON 的扩展）：BSON（Binary Serialized dOcument Format）存储形式是指：存储在集合中的文档，被存储为键-值对的形式。键用于唯一标识一个文档，为字符串类型，而值则可以是各中复杂的文件类型。
- 可通过网络访问：MongoDB 服务端可运行在 Linux、Windows 或 OS X 平台，支持 32 位和 64 位应用，默认端口为 27017。推荐运行在 64 位平台，因为 MongoDB 在 32 位模式运行时支持的最大文件尺寸为 2GB。MongoDB 把数据存储在文件中（默认路径为：/data/db），为提高效率使用内存映射文件进行管理。

10.10.3 Hypertable

Hypertable 是一个开源、高性能、可伸缩的数据库，它采用与 Google 的 Bigtable 相似的模型。在过去数年中，Google 为在 PC 集群 上运行的可伸缩计算基础设施设计建造了三个关键部分。第一个关键的基础设施是 Google File System（GFS），这是一个高可用的文件系统，提供了一个全局的命名空间。它通过跨机器（和跨机架）的文件数据复制来达到高可用性，并因此免受传统 文件存储系统无法避免的许多失败的影响，比如电源、内存和网络端口等失败。第二个基础设施是名为 Map-Reduce 的计算框架，它与 GFS 紧密协作，帮助处理收集到的海量数据。第三个基础设施是 Bigtable，它是传统数据库的替代。Bigtable 让你可以通过一些主键来组织海量数据，并实现高效的查询。Hypertable 是 Bigtable 的一个开源实现，并且根据我们的想法进行了一些改进。

10.10.4 Apache Cassandra

Apache Cassandra 是一套开源分布式 Key-Value 存储系统。它最初由 Facebook 开发，用

于储存特别大的数据。Facebook 目前在使用此系统。

Cassandra 的主要特性包括以下几个方面：

- 分布式；
- 基于列的结构化；
- 高伸展性。

Cassandra 的主要特点就是它不是一个数据库，而是由一堆数据库节点共同构成的一个分布式网络服务，对 Cassandra 的一个写操作，会被复制到其他节点上去，对 Cassandra 的读操作，也会被路由到某个节点上面去读取。对于一个 Cassandra 群集来说，扩展性能是比较简单的事情，只管在群集里面添加节点就可以了。

Cassandra 是一个混合型的非关系的数据库，类似于 Google 的 BigTable。其主要功能比 Dynomite（分布式的 Key-Value 存储系统）更丰富，但支持度却不如文档存储 MongoDB（介于关系数据库和非关系数据库之间的开源产品，是非关系数据库当中功能最丰富，最像关系数据库的。支持的数据结构非常松散，是类似 json 的 bjson 格式，因此可以存储比较复杂的数据类型。）Cassandra 最初由 Facebook 开发，后转变成了开源项目。它是一个网络社交云计算方面理想的数据库。以 Amazon 专有的完全分布式的 Dynamo 为基础，结合了 Google BigTable 基于列族（Column Family）的数据模型。P2P 去中心化的存储。很多方面都可以称之为 Dynamo 2.0。

和其他数据库比较，Cassandra 的突出特点是：

- **模式灵活**：使用 Cassandra，像文档存储，你不必提前解决记录中的字段。你可以在系统运行时随意的添加或移除字段。这是一个惊人的效率提升，特别是在大型部署上。
- **真正的可扩展性**：Cassandra 是纯粹意义上的水平扩展。为给集群添加更多容量，可以指向另一台电脑。你不必重启任何进程，改变应用查询，或手动迁移任何数据。
- **多数据中心识别**：你可以调整你的节点布局来避免某一个数据中心起火，一个备用的数据中心将至少有每条记录的完全复制。
- **范围查询**：如果你不喜欢全部的键值查询，则可以设置键的范围来查询。
- **列表数据结构**：在混合模式可以将超级列添加到 5 维。对于每个用户的索引，这是非常方便的。
- **分布式写操作**：有可以在任何地方任何时间集中读或写任何数据。并且不会有任何单点失败。

本章小结

本章首先介绍了 NoSQL 概念以及发展现状，阐述了为什么要使用 NoSQL 数据库以及 NoSQL 数据库的特点；然后，介绍了 NoSQL 数据库的五大挑战及其面临的质疑；接下来，介绍了 NoSQL 的三大基石，即 CAP、BASE 和最终一致性；接下来对 NoSQL 数据库和关系数据库进行了简单比较；最后，给出了典型的 NoSQL 数据库分类和开源软件。

参考文献

[1] 颜开. NoSQL 数据库笔谈. 百度文库. http://wenku.baidu.com/view/246e00d4195f312b3169a570.html

[2] CAP 原理. 百度文库. http://wenku.baidu.com/view/7f25f00d7cd184254b353530.html

[3] 孟小峰, 慈祥. 大数据管理：概念、技术与挑战. 计算机学报, 2013 年第 8 期.

第 11 章 云数据库

云数据库是在 SaaS（Software-as-a-Service：软件即服务）成为应用趋势的大背景下发展起来的云计算技术，它极大地增强了数据库的存储能力，消除了人员、硬件、软件的重复配置，让软、硬件升级变得更加容易，同时，也虚拟化了许多后端功能。云数据库具有高可扩展性、高可用性、采用多租形式和支持资源有效分发等特点。可以说，云数据库是数据库技术的未来发展方向。

本章介绍云数据库的相关知识，内容要点如下：

- 云数据库概述
- 云数据库的特性
- 云数据库是海量存储需求的必然选择
- 云数据库与传统的分布式数据库
- 云数据库的影响
- 云数据库产品
- 数据模型
- 数据访问方法
- 编程模型

11.1 云数据库概述

11.1.1 云计算和 SaaS

云计算（Cloud Computing）是分布式计算（Distributed Computing）、并行计算（Parallel Computing）、效用计算（Utility Computing）、网络存储（Network Storage Technologies）、虚拟化（Virtualization）、负载均衡（Load Balance）等传统计算机和网络技术发展融合的产物。

云计算由一系列可以动态升级和被虚拟化的资源组成，这些资源被所有云计算的用户共享并且可以方便地通过网络访问，用户无需掌握云计算的技术，只需要按照个人或者团体的需要租赁云计算的资源。云计算是继 1980 年代大型计算机到客户端-服务器的大转变之后的又一种巨变。云计算的出现并非偶然，早在上世纪 60 年代，麦肯锡就提出了把计算能力作

为一种像水和电一样的公用事业提供给用户的理念，这成为云计算思想的起源。在 20 世纪 80 年代网格计算，90 年代公用计算，21 世纪初虚拟化技术、SOA、SaaS 应用的支撑下，云计算作为一种新兴的资源使用和交付模式已经被学界和产业界所广泛认知和接受。中国云发展创新产业联盟评价云计算为“信息时代商业模式上的创新”。

云计算包括三种主要类型，即 IaaS（Infrastructure as a Service）、PaaS（Platform as a Service）和 SaaS（Software as a Service）：

- **IaaS（Infrastructure as a Service）**

IaaS（Infrastructure as a Service），即“基础设施即服务”。提供给消费者的服务是对所有设施的利用，包括处理、存储、网络和其它基本的计算资源，用户能够部署和运行任意软件，包括操作系统和应用程序。消费者不管理或控制任何云计算基础设施，但能控制操作系统的选择、储存空间、部署的应用，也有可能获得有限制的网络组件（例如，防火墙、负载均衡器等）的控制。

在没有 IaaS 之前，如果你想在办公室或者公司的网站上运行一些企业应用，你需要去买服务器或者别的昂贵的硬件来控制本地应用，让你的业务运行起来。但是，现在有了 IaaS，你可以将硬件外包到别的地方去。IaaS 公司会提供场外服务器、存储和网络硬件，你可以租用这些基础设施，从而节省了维护成本和办公场地，公司可以在任何时候利用这些硬件来运行其应用。一些大的 IaaS 公司包括 Amazon、Microsoft、VMWare、Rackspace 和 Red Hat。

- **PaaS（Platform as a Service）**

PaaS（Platform as a Service），即“平台即服务”。提供给消费者的服务是，把客户采用所提供的开发语言和工具（例如 Java，python, .Net 等）开发的、或收购的应用程序部署到供应商的云计算基础设施上去。客户不需要管理或控制底层的云基础设施，包括网络、服务器、操作系统、存储等，但客户能控制部署的应用程序，也可能控制运行应用程序的托管环境配置。

你公司所有的开发都可以在这一层进行，节省了时间和资源。PaaS 公司在网上提供各种开发和分发应用的解决方案，比如虚拟服务器和操作系统。这节省了你在硬件上的费用，也让分散的工作室之间的合作变得更加容易。一些大的 PaaS 提供者有 Google App Engine、Microsoft Azure、Force.com、Heroku、Engine Yard。最近兴起的公司有 AppFog、 Mendix 和 Standing Cloud。

- **SaaS（Software as a Service）**

SaaS（Software as a Service），即“软件即服务”。它是一种通过 Internet 提供软件的模式，厂商将应用软件统一部署在自己的服务器上，客户可以根据自己实际需求，通过互联网向厂商定购所需的应用软件服务，按定购的服务多少和时间长短向厂商支付费用，并通过互联网获得厂商提供的服务。用户不用再购买软件，而改用向提供商租用基于 Web 的软件，来管理企业经营活动，且无需对软件进行维护，服务提供商会全权管理和维护软件。对于许多小型企业来说，SaaS 是采用先进技术的最好途径，它消除了企业购买、构建和维护基础设施和应用程序的需要。

在这种模式下，客户不再像传统模式那样花费大量投资用于硬件、软件、人员，而只需要支出一定的租赁服务费用，通过互联网便可以享受到相应的硬件、软件和维护服务，享有软件使用权和不断升级服务；公司上项目不用再像传统模式一样需要大量的时间用于布置系统，多数经过简单的配置就可以使用。这是网络应用最具效益的营运模式。

Salesforce 是 SaaS 厂商的先驱，它一开始提供的是可通过网络访问的销售力量自动化应用软件。在该公司的带动下，其他 SaaS 厂商已如雨后春笋般蓬勃而起。

11.1.2 云数据库概念

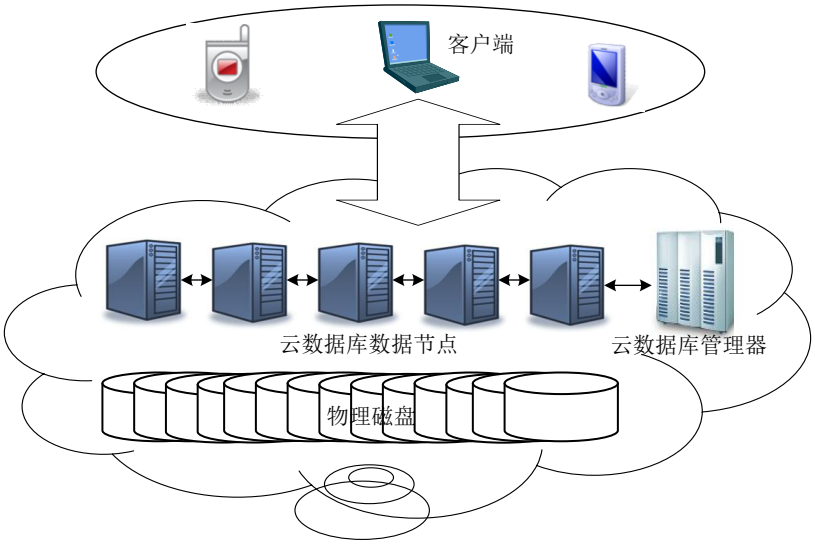

图 11-1 云数据库应用示意图

云数据库是在 SaaS（Software-as-a-Service：软件即服务）成为应用趋势的大背景下发展起来的云计算技术，它极大地增强了数据库的存储能力，消除了人员、硬件、软件的重复配

置，让软、硬件升级变得更加容易，同时，也虚拟化了许多后端功能。云数据库具有高可扩展性、高可用性、采用多租形式和支持资源有效分发等特点。可以说，云数据库是数据库技术的未来发展方向。目前，对于云数据库的概念界定不尽相同，本文采用的云数据库定义是：云数据库是部署和虚拟化在云计算环境中的数据库。

如图 11-1 所示，在云数据库应用中，客户端不需要了解云数据库的底层细节，所有的底层硬件都已经被虚拟化，对客户端而言是透明的，它就像在使用一个运行在单一服务器上的数据库一样，非常方便容易，同时又可以获得理论上近乎无限的存储和处理能力。

11.2 云数据库的特性

云数据库具有以下特性：

（1）**动态可扩展**：理论上，云数据库具有无限可扩展性，可以满足不断增加的数据存储需求。在面对不断变化的条件时，云数据库可以表现出很好的弹性。例如，对于一个从事产品零售的电子商务公司，会存在季节性或突发性的产品需求变化，或者对于类似 Animoto 的网络社区站点，可能会经历一个指数级的增长阶段，这时，就可以分配额外的数据库存储资源来处理增加的需求，这个过程只需要几分钟。一旦需求过去以后，就可以立即释放这些资源。

（2）**高可用性**：不存在单点失效问题。如果一个节点失效了，剩余的节点就会接管未完成的事务。而且，在云数据库中，数据通常是复制的，在地理上也是分布的。诸如 Google、Amazon 和 IBM 等大型云计算供应商，具有分布在世界范围内的数据中心，通过在不同地理区间内进行数据复制，可以提供高水平的容错能力。例如，Amazon SimpleDB 会在不同的区间内进行数据复制，因此，即使整个区域内的云设施发生失效，也可以保证数据继续可用。

（3）**较低的使用代价**：通常采用多租户（multi-tenancy）的形式，这种共享资源的形式对于用户而言可以节省开销；而且用户采用按需付费的方式使用云计算环境中的各种软、硬件资源，不会产生不必要的资源浪费。另外，云数据库底层存储通常采用大量廉价的商业服务器，这也大大降低了用户开销。

（4）**易用性**：使用云数据库的用户不用控制运行原始数据库的机器，也不必了解它身在何处。用户只需要一个有效的连接字符串就可以开始使用云数据库。

（5）**大规模并行处理**：支持几乎实时的面向用户的应用、科学应用和新类型的商务解

决方案。

11.3 云数据库是海量存储需求的必然选择

云数据库在当前数据爆炸的大数据时代具有广阔的应用前景。根据 IDC 的研究报告，在未来的 5 年中，企业对结构化数据的存储需求会每年增加 20%左右，而对非结构化数据的存储需求将会每年增加 60%左右。在小规模应用的情况下，系统负载的变化可以由系统空闲的多余资源来处理，但是，在大规模应用的情况下，不仅存在海量的数据存储需求，而且应用对资源的需求也是动态变化的，这意味着大量虚拟机的增加或减少。对于这种情形，传统的关系数据库已经无法满足要求，云数据库成为必然的选择。换句话说，大数据时代的海量存储需求催生了云数据库。

11.4 云数据库与传统的分布式数据库

分布式数据库是计算机网络环境中各场地或节点上的数据库的逻辑集合。逻辑上它们属于同一系统，而物理上它们分散在用计算机网络连接的多个节点，并统一由一个分布式数据库管理系统管理。

分布式数据库已经存在很多年，它可以用来管理大量的分布存储的数据，并且通常采用非共享的体系架构。云数据库和传统的分布式数据库有着相似的地方，比如，都把数据存放到不同的节点上。但是，分布式数据库在可扩展性方面是无法和云数据库相比的。由于需要考虑数据同步和分区失败等开销，前者随着节点的增加，会导致性能快速下降。而后者则具有很好的可扩展性，因为后者在设计的时候，就已经避免了许多会影响到可扩展性的因素，比如采用更加简单的数据模型、对元数据和应用数据进行分离以及放松对一致性的要求等等。另外，在使用方式上，云数据库也不同于传统的分布式数据库，云数据库通常采用多租户模式，即多个租户共用一个实例，租户的数据既有隔离又有共享，从而解决数据存储的问题，同时也降低了用户使用数据库的成本。

11.5 云数据库的影响

云数据库的影响主要体现在以下几个方面：

（1）**极大地改变企业管理数据的方式**。Forrester Research 分析师 Noel Yuhanna 指出，

18%的企业正在把目光投向云数据库。对于中小企业而言，云数据库可以允许他们在 WEB 上快速搭建各类数据库应用，越来越多的本地数据和服务将逐渐被转移到云中。企业用户可以在任意地点通过简单的终端设备，就可以对企业数据进行全面管理。此外，云数据库可以很好地支持企业开展一些短期项目，降低开销，而不需要企业为某个项目单独建立昂贵的数据中心。但是，云数据库的成熟仍然需要一段时间。中小企业会更多地采用云数据库产品，但是，对于大企业而言，云数据库并非首选，因为大企业通常自己建造数据中心。

（2）**催生新一代的数据库技术**。IDC 的数据库分析师 Carl Olofson 认为，云模型提供了无限的处理能力以及大量的 RAM，因此，云模型将会极大改变数据库的设计方式，将会出现第三代数据库技术。第一代是 20 世纪 70 年代的早期关系数据库，第二代是 80 到 90 年代的更加先进的关系模型。第三代的数据库技术，要求数据库能够灵活处理各种类型的数据，而不是强制让数据去适应预先定制的数据结构。事实上，从目前云数据库产品中的数据模型设计方式来看，已经有些产品（比如 SimpleDB、HBase、Dynamo、BigTable）放弃传统的行存储方式，而采用键/值存储，从而可以在分布式的云环境中获得更好的性能。可以预期的是，云数据库将会吸引越来越多的学术界的目光，该领域的相关问题也将成为未来一段时间内数据库研究的重点内容，比如云数据库的体系架构和数据模型等等。

（3）**数据库市场份额面临重新分配**。在过去的几十年里，数据库市场一直被诸如 Teradata、Oracle、IBM DB2、Microsoft SQL Server、Sybase 等传统数据库厂商所垄断。随着云数据库的出现和不断发展，市场将面临重新洗牌。首先，Amazon 和 Google 等原本并不从事数据库业务的国际知名企业，也乘着云计算的东风，开发了云中的数据库产品，加入这场新兴市场的角逐。实际上，对于云数据库市场而言，Amazon SimpleDB 和 Google BigTable 这类产品扮演了引领者的角色，传统的数据库厂商已然成为跟进者。其次，一些新的云数据库厂商开始出现，并且推出了具有影响力的产品，比如 Vertica 的 Analytic Database for the Cloud 和 EnterpriseDB 的 Postgres Plus in the Cloud。因此，数据库市场份额的重新分配不可避免。

11.6 云数据库产品

云数据库供应商主要分为三类：

- **传统的数据库厂商**：Teradata、Oracle、IBM DB2 和 Microsoft SQL Server；

- **涉足数据库市场的云供应商**：Amazon、Google 和 Yahoo；
- **新兴小公司**：Vertica、LongJump 和 EnterpriseDB。

就目前阶段而言，虽然一些云数据库产品，比如 Google BigTable、SimpleDB 和 HBase，在一定程度上实现了对于海量数据的管理，但是，这些系统暂时还不完善，只是“云数据库”的雏形。让这些系统支持更加丰富的操作以及更加完善的数据管理功能（比如复杂查询和事务处理），以满足更加丰富的应用，仍然需要研究人员的不断努力。

表 11-1 给出了市场上常见的云数据库产品，对于其中一些主要产品，下面我们会做简要介绍。

表 11-1 云数据库产品

企业	产品
Amazon	Dynamo、SimpleDB、RDS
Google	BigTable、FusionTable
Microsoft	Microsoft SQL Server Data Services 或 SQL Azure
Oracle	Oracle Cloud
Yahoo!	PNUTS
Vertica	Analytic Database v3.0 for the Cloud
EnerpriseDB	Postgres Plus in the Cloud
开源项目	HBase、Hypertable
其他	EnerpriseDB、FathomDB、ScaleDB、Objectivity/DB、M/DB:X

11.6.1 Amazon 的云数据库产品

Amazon 是云数据库市场的先行者。Amazon 除了提供著名的 S3 存储服务和 EC2 计算服务以外，还提供基于云的数据库服务 Dynamo。Dynamo 采用“键/值”存储，其所存储的数据是非结构化数据，不识别任何结构化数据，需要用户自己完成对值的解析。Dynamo 系统中的键（key）不是以字符串的方式进行存储，而是采用 md5_key（通过 md5 算法转换后得到）的方式进行存储，因此，它只能根据 key 去访问，不支持查询。SimpleDB 是 Amazon 公司

开发的一个可供查询的分布数据存储系统，它是 Dynamo“键/值”存储的补充和丰富。顾名思义，SimpleDB 的目的是作为一个简单的数据库来使用，它的存储元素（属性和值）是由一个 *id* 字段来确定行的位置。这种结构可以满足用户基本的读、写和查询功能。SimpleDB 提供易用的 API 来快速地存储和访问数据。但是，SimpleDB 不是一个关系型数据库，传统的关系型数据库采用行存储，而 SimpleDB 采用了“键/值”存储，它主要是服务于那些不需要关系数据库的 WEB 开发者。

Amazon RDS（Amazon Relational Database Service）是 Amazon 开发的一种 Web 服务，它可以让用户在云环境中建立、操作关系型数据库（目前支持 MySQL 和 Oracle 数据库）。用户只需要关注应用和业务层面的内容，而不需要在繁琐的数据库管理工作上耗费过多的时间。

此外，Amazon 和其他数据库厂商开展了很好的合作，Amazon EC2 应用托管服务已经可以部署很多种数据库产品，包括 SQL Server、Oracle 11g、MySQL 和 IBM DB2 等主流数据库平台，以及其他一些数据库产品，比如 EnerpriseDB。作为一种可扩展的托管环境，开发者可以在 EC2 环境中开发并托管自己的数据库应用。

11.6.2 Google 的云数据库产品

Google BigTable 是一种满足弱一致性要求的大规模数据库系统。Google 设计 BigTable 的目的，是为了处理 Google 内部大量的格式化及半格式化数据。目前，许多 Google 应用都是建立在 BigTable 之上的，比如 WEB 索引、Google Earth、Google Finance、Google Maps 和 Search History。BigTable 是构建在其他几个 Google 基础设施之上的。首先，BigTable 使用了分布式 Google 文件系统 GFS（Google File System）来存储日志和数据文件；其次，BigTable 依赖一个高可用的、持久性的分布式锁服务 Chubby；再次，BigTable 依赖一个簇管理系统来调度作业、在共享机器上调度资源、处理机器失败和监督机器状态。

但是，和 Amazon SimpleDB 类似，就目前而言，BigTable 实际上还不是真正的 DBMS（Database Management System），它无法提供事务一致性、数据一致性。这些产品基本上可以被看成是云环境中的表单。

Google Cloud SQL 是谷歌公司推出的基于 MySQL 的云数据库，使用 Cloud SQL 的好处显而易见，所有的事务都在云中，并由谷歌管理，用户不需要配置或者排查错误，仅仅依靠

它来开展工作即可。由于数据在谷歌多个数据中心中复制，因此它永远是可用的。谷歌还将提供导入或导出服务，方便用户将数据库带进或带出云。谷歌使用用户非常熟悉的 MySQL，因此多数应用程序不需过多调试即可运行，数据格式对于大多数开发者和管理员来说也是非常熟悉的。还有一个好处就是与应用引擎集成。

11.6.3 Microsoft 的云数据库产品

2008 年 3 月，微软通过 SQL Data Service（SDS）提供 SQL Server 的 RDBMS 功能，这使得微软成为云数据库市场上的第一个大型数据库厂商。此后，微软对 SDS 功能进行了扩充，并且重新命名为 SQL Azure。微软的 Azure 平台提供了一个 WEB 服务集合，可以允许用户通过网络在云中创建、查询和使用 SQL Server 数据库，云中的 SQL Server 服务器的位置对于用户而言是透明的。对于云计算而言，这是一个重要的里程碑。SQL Azure 具有以下特性：

- 属于关系型数据库：支持使用 TSQL（Transact Structured Query Language）来管理、创建和操作云数据库；
- 支持存储过程：它的数据类型、存储过程和传统的 SQL Server 具有很大的相似性，因此，应用可以在本地进行开发，然后部署到云平台上；
- 支持大量数据类型：包含了几乎所有典型的 SQL Server 2008 的数据类型；
- 支持云中的事务：支持局部事务，但是不支持分布式事务。

11.6.4 开源云数据库产品

HBase 和 Hypertable 利用开源 MapReduce 平台 Hadoop 提供了类似于 BigTable 的可伸缩数据库实现。MapReduce 是 Google 开发的、用来运行大规模并行计算的框架。采用 MapReduce 的应用更像一个人提交的批处理作业，但是，这个批处理作业不是在单个服务器上运行，应用和数据都是分布在多个服务器上。Hadoop 是由 Yahoo 资助的一个开源项目，是 MapReduce 的开源实现，从本质上来说，它提供了一个使用大量节点来处理大规模数据集的方式。

HBase 已经成为 Apache Hadoop 项目的重要组成部分，并且已经在生产系统中得到应用。

与 HBase 类似的是 Hypertable。不过，HBase 的开发语言是 Java，而 Hypertable 则采用 C/C++开发。与 HBase 相比，Hypertable 具有更高的性能。但是，HBase 不支持 SQL（Structual Query Language）类型的查询语言。

甲骨文开源数据库产品 BerkelyDB 也提供了云计算环境中的实现。

11.6.5 其他云数据库产品

Yahoo! PNUTS 是一个为网页应用开发的、大规模并行的、地理分布的数据库系统，它是 Yahoo!云计算平台重要的一部分。Vertica Systems 在 2008 年发布了云版本的数据库。10Gen 公司的 Mongo、AppJet 的 AppJet 数据库也都提供了相应的云数据库版本。M/DB:X 是一种云中的 XML 数据库，它通过 HTTP/REST 访问。FathomDB 旨在满足基于 Web 的公司提出的高传输要求，它所提供的服务更倾向于在线事务处理而不是在线分析处理。IBM 投资的 EnerpriseDB 也提供了一个运行在 Amazon EC2 上的云版本。LongJump 是一个与 Salesforce。com 竞争的新公司，它推出了基于开源数据库 PostgreSQL 的云数据库产品。Intuit QuickBase 也提供了自己的云数据库系列。麻省理工学院研制的 Relational Cloud 可以自动区分负载的类型，并把类型近似的负载分配到同一个数据节点上，而且采用了基于图的数据分区策略，对于复杂的事务型负载也具有很好的可扩展性，此外，它还支持在加密的数据上运行 SQL 查询。

11.7 数据模型

云数据库的设计可以采用不同的数据模型，不同的数据模型可以满足不用应用类型的需求，主要包括：键/值模型和关系模型。

11.7.1 键/值模型

BigTable、Dynamo、SimpleDB、PNUTS、HBase 等产品都采用了键/值模型存储数据。下面我们以 Google BigTable 的数据模型为例来介绍键/值模型。

Google BigTable 的数据模型：BigTable 和它的同类开源产品 HBase，提供了一个不同于以往的简单、动态的、非关系型的数据模型。BigTable 采用了键/值数据模型。在 BigTable 中，包括行列以及相应的时间戳在内的所有数据都存放在表格的单元里。BigTable 的内容按

照行来划分，多个行组成一个小表（Tablet），保存到某一个服务器节点中，这就意味着，每个 Tablet 包含了位于某个区间内的所有数据。对于 BigTable 而言，一个数据簇中存储了许多表，其中，每个表都是一个 Tablet 集合。在最初阶段，每个表只包含一个 Tablet。随着表的增长，它会被自动分解成许多 Tablet，每个 Tablet 默认尺寸大约是 100 到 200MB。BigTable 使用一个类似于 B+树的三层架构来存储 Tablet 位置信息。由于 BigTable 采用了键/值数据模型，因此，不存在表间的联接操作，这也使得数据分区操作相对简单，只需要根据键的区间来划分即可。

一个 BigTable 实际上就是一个稀疏的、分布的、永久的多维排序图，它采用行键盘（row key）、列键（column key）和时间戳（timestamp）对图进行索引。图中的每个值都是未经解释的字节数组：

- **行键**：BigTable 在行键上根据字典顺序对数据进行维护。对于一个表而言，行区间是根据行键的值进行动态划分的。每个行区间称为一个 Tablet，它是负载均衡和数据分发的基本单位，这些 Tablet 会被分发到不同的数据服务器上。
- **列键**：被分组成许多“列家族”的集合，它是基本的访问控制单元。存储在一个列家族当中的所有数据，通常都属于同一种数据类型，这通常意味着具有更高的压缩率。数据可以被存放到列家族的某个列键下面，但是，在把数据存放到这个列家族的某个列键下面之前，必须首先创建这个列家族。在创建完成一个列家族以后，就可以使用同一个家族当中的列键。
- **时间戳**：在 BigTable 中的每个单元格当中，都包含相同数据的多个版本，这些版本采用时间戳进行索引。BitTable 时间戳是 64 位整数。一个单元格的不同版本是根据时间戳降序的顺序进行存储的，这样，最新的版本可以被最先读取。

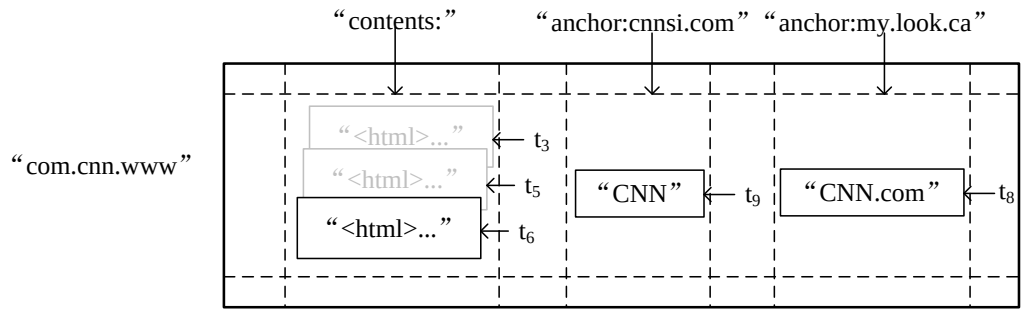

图 11-2 BigTable 数据模型的一个实例

这里以一个实例来阐释 BigTable 的数据模型。图 11-2 显示了存储了网页数据的 WebTable 的一个片段。行名称是反转的 URL，*contents* 列家族包含了网页内容，*anchor* 列家族包含了任何引用这个页面的 anchor 文本。CNN 的主页被 Sports Illustrated 和 MY-look 主页同时引用，因此，这里的行包含了名称为”anchor:cnnsi.com”和”anchor:my.look.ca”的列。每个 *anchor* 单元格都只有一个版本，*contents* 列有三个版本，分别对应于时间戳 t_3，t_5 和 t_6。

HBase 和 BigTable 一样，也采用了键/值数据模型，它是一个开放源码、分布式、面向列、多维、高可用、高性能的存储技术，采用 JAVA 语言编写。作为一个使用了 Hadoop 的分布式数据库，HBase 可以实现结构化数据的可靠存储。就像 Google BigTable 充分利用 Google File System 提供的分布式数据存储功能一样，HBase 的目的就是在 HDFS（Hadoop Distributed File System）之上提供类似 BigTable 的功能。HBase 采用多层索引表来执行键/值映射，获得了优越的主键查询性能。

HBase、BigTable 中的数据库模式和关系型模式具有很大的区别。第一，不存在表间的联接操作；第二，整个模式也只有一个索引——行键。与 RDBMS（Relational Database Management System）不同的是，开发者不需要使用 WHERE 从句的等价表达形式。通过设计，HBase 中的所有访问方法，或者通过行键访问，或者通过行键扫描，从而使得整个系统不会慢下来。由于 HBase 位于 Hadoop 框架之上，MapReduce 就可以用来生成索引表。

HBase 列家族可以被配置成支持不同类型的访问模式。一个家族也可以被设置成放入内存当中，以消耗内存为代价，从而换取更好的响应性能。

此外，Amazon Dynamo、SimpleDB 也都和 BigTable 一样采用了键/值存储。SimpleDB 中包含三个概念：domain、item 和 attribute，其中，domain 相当于一个 table，item 相当于一行，attribute 相当于一列，一列可以有多个值。但是，SimpleDB 和 BigTable 在数据划分的方式上存在一些差别：

- **SimpleDB 的数据划分**：采用静态数据划分方法，利用哈希函数把数据分发到多个数据节点，这使得 SimpleDB 更像一个哈希系统。这种方法的优点是，实现难度小；但是缺点也很明显，即用户的一些 domain 存放不连续。为了降低不连续数据存放对用户查询性能带来的负面影响，SimpleDB 对用户 domain 的大小进行限制，比如一个 domain 大小不超过 10GB。
- **BigTable 的数据划分**：采用动态划分方法，一个用户的数据可能会被划分成多个

Tablet，分发到不同的数据节点上。这种方法的优点是可以很好地支持负载均衡，缺点是需要相关的 Tablet 分拆与合并机制。

BigTable、Dynamo、SimpleDB、PNUTS、HBase 等产品虽然都采用了键/值模型存储数据，但是，这些系统在进行数据访问的时候都是以单个键作为访问粒度，这种方式对于一些网页应用而言无法取得好的性能，比如在线游戏、社区网络、合作编辑等包含合作性质的应用，这些应用需要对一个键组(key group)进行一致性的访问。由此，名为 G-Store 的键/值系统，并提出了“键组协议”来对一组键进行一致的、高效的访问，也就是说，在该系统中，数据访问的粒度不再是单个的键，而是一个键组。

11.7.2 关系模型

微软的 SQL Azure 云数据库采用了关系模型，它的数据分区方式和 BigTable 有些不同。关系型云数据库的数据模型涉及行组和表组等相关概念。

一个表是一个逻辑关系，它包含一个*分区键*，用来对表进行分区。具有相同分区键的多个表的集合，称为*表组*。在表组中，具有相同分区键值的多个行的集合，称为*行组*。一个行组中包含的行，总是被分配到同一个数据节点上。每个表组会包含多个行组，这些行组会被分配到不同的数据节点上。一个数据分区包含了多个行组。因此，每个数据节点都存储了位于某个分区键值区间内的所有行。

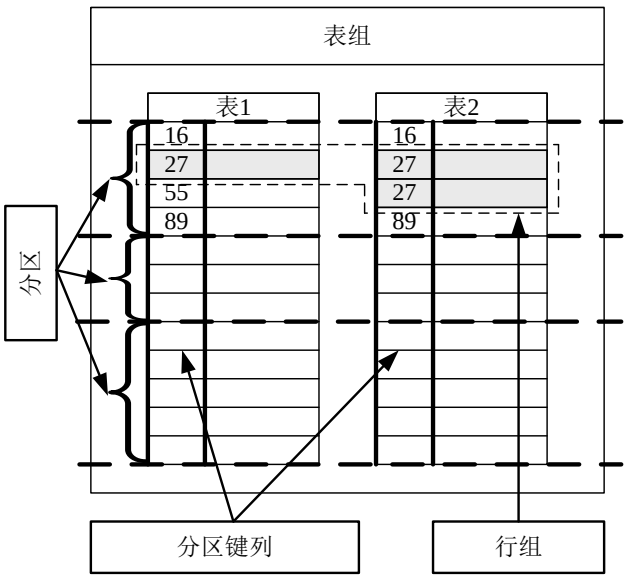

图 11-3 关系型云数据库的数据模型

这里以一个实例来解释关系型云数据库的数据模型。如图 11-3 所示，一个表组包含了两个相关的表，即图 11-3 中的表 1 和表 2。表 1 和表 2 分别包含了一个分区键列，每个分区键列是由多个分区键值组成的，这两个分区键列具有相同类型的分区键，都是 *ID* 类型的数值型数据。表 1 中的 *ID* 列和表 2 中的 *ID* 列存在主外键关联。因此，表 1 中和表 2 中的 *ID* 列值为 27 的三个行（图中用灰色底纹表示，并且用虚线框包围的部分），就属于一个行组。从图中也可以看出，表组被分割成多个分区。而且，同一行组中的内容都位于同一个数据分区内。

一个表组通常包含那些会被某个应用同时使用的多个表，因此，需要把这些表存储在一起，从而加快查询效率。比如，有两个表 *STUDENT_INFO* 和 *BOOKBORROWING_INFO*，前者记录的信息包括名字、性别、年龄、电话等等，而后者则记录了借书的相关信息。当图书馆管理员查询一个人的借书记录时，通常需要同时查看借阅者姓名、年龄、电话、借书数量、借书时间等信息，这就需要在 *STUDENT_INFO* 和 *BOOKBORROWING_INFO* 这两个表之间进行联接操作。很先让，如果把这两个表存储在同一个数据节点上，联接操作会快很多。

类似地，把一个行组中的多个行存储在一个数据分区内，也具有明显的好处。比如，图 11-3 中的表 1 和表 2 中 ID 为 27 的的多个行构成一个行组，它们之间存在主外键关联，把它们存放在一起可以加快查询的效率。

11.8 数据访问方法

这里以一个实例来阐释云数据库的数据访问方法。如图 11-4 所示，当客户端请求数据时，它首先向管理器请求一份分区映射图，管理器向客户端发送分区映射图，客户端收到以后，在图中进行搜寻，根据键值找到自己所需数据的存储位置，然后客户端到指定的数据节点请求数据，最后，由该数据节点把数据返回给客户端。实际上，为了改进性能，同时也为了避免管理器的性能瓶颈，通常会在客户端缓存常用的分区映射图，这样，客户端在很多情况下不用与管理器交互就可以直接访问相应的数据节点。

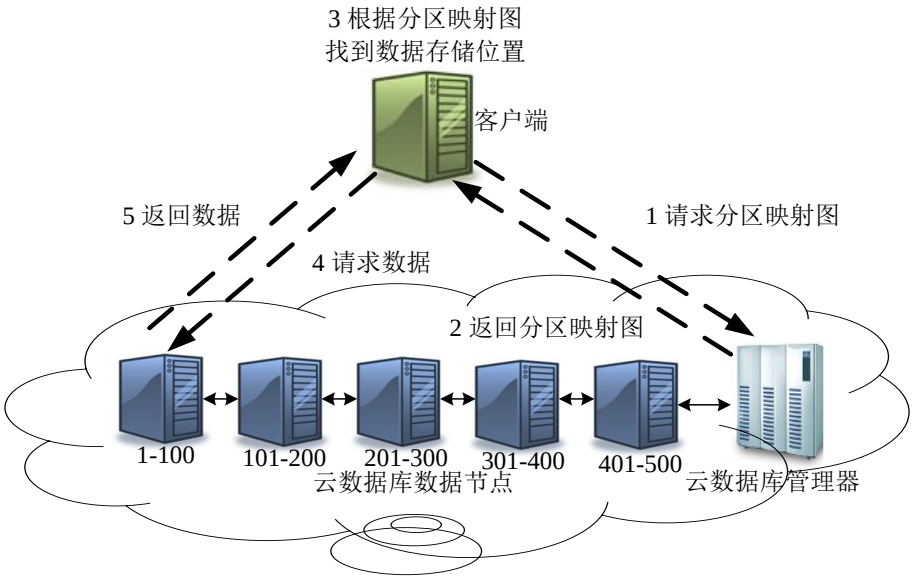

图 11-4 云数据库中的数据访问方法

11.9 编程模型

云数据库存储了海量数据，涉及到大量的数据运算，如果仍然采用传统的数据处理方法，将无法充分发挥云环境的优点。因此，采用一种机制简单而又具有高可扩展性的编程方法就显得尤为重要。查询这种大型的数据集，可以采用 Google 发明的一种新的编程模型，称为 MapReduce。

MapReduce 编程模型用来处理跨越成百上千个机器的大规模数据集，该软件架构隐藏了并行计算、数据分布和失败处理的细节。它背后的思想是，在分布式环境下处理数据，总是包括两个基本步骤：(1)映射所需的数据；（2）对这些数据进行聚集（aggregate）操作。它让用户定义一个映射函数，这个映射函数把一个“key/value”对的集合转换成一个中间临时的“key/value”对的集合，然后使用一个 reduce 函数，把所有那些与同一个键相关的中间临时值都进行合并。许多现实世界的任务都可以采用这个模型来表达。MapReduce 可以运行在大规模商用服务器簇上，并且具有很高的可扩展性。一个典型的 MapReduce 计算会在成千上万的机器上处理许多 TB 的数据。

但是，传统的关系型数据库和 MapReduce 对数据的处理方式存在很大的区别，因此，需要把关系数据库中的一些操作转换成 MapReduce 操作。这其中一类典型的操作就是联接(join)操作。

总的来说，在 MapReduce 环境下执行两个关系的联接操作的方法如下：假设关系 R(A，

B)和 S(B,C)都存储在一个文件中。为了联接这些关系，必须把来自每个关系的各个元组都和一个 key 关联，这个 key 就是属性 B 的值。可以使用一个 Map 进程集合，把来自 R 的每个元组(a,b)转换成一个 key-value 对，其中的 key 就是 b，值就是(a,R)。注意，这里把关系 R 包含到 value 中，这样做使得我们可以在 Reduce 阶段，只把那些来自 R 的元组和来自 S 的元组进行匹配。类似地，可以使用一个 Map 进程集合，把来自 S 的每个元组(b,c)，转换成一个 key-value 对，key 是 b，value 是(c,S)。这里把关系名字包含在属性值中，可以使得在 Reduce 阶段只把那些来自不同关系的元组进行合并。Reduce 进程的任务就是，把来自关系 R 和 S 的具有共同属性 B 值的元组进行合并。这样，所有具有特定 B 值的元组必须被发送到同一个 Reduce 进程。假设使用 k 个 Reduce 进程。这里选择一个哈希函数 h，它可以把属性 B 的值映射到 k 个哈希桶，每个哈希值对应一个 Reduce 进程。每个 Map 进程把 key 是 b 的 key-value 对，都发送到与哈希值 h(b)对应的 Reduce 进程。Reduce 进程把联接后的元组(a,b,c)，写到一个单独的输出文件中。

本章小结

本章首先介绍了云数据库的概念和特性，然后指出云数据库是海量存储需求的必然选择；接下来，比较了云数据库和传统的分布式数据库，并阐述了云数据库的影响；然后，介绍了具有代表性的云数据库产品，包括 Amazon、Google 和 Microsoft 的产品；最后，介绍了云数据库的数据模型和数据访问方法。

参考文献

[1] 林子雨,赖永炫,林琛,谢怡,邹权. 云数据库研究. 软件学报.2012,23(5):1148-1166.

（注：本章内容全部来自林子雨发表的《软件学报》论文“云数据库研究”）

第 12 章 Google Spanner

Spanner 是一个可扩展、多版本、全球分布式并且支持同步复制的数据库，它是 Google 的第一个可以全球扩展并且支持外部一致性的数据库。Spanner 能做到这些，离不开一个用 GPS 和原子钟实现的时间 API。这个 API 能将数据中心之间的时间同步精确到 10ms 以内。因此，Spanner 有几个给力的功能：无锁读事务、原子模式修改、读历史数据无阻塞。

本章介绍 Google Spanner 相关知识，内容要点如下：

- Spanner 背景
- 与 BigTable、Megastore 的对比
- Spanner 的功能
- 体系结构
- Spanserver
- Directory
- 数据模型
- TrueTime
- Spanner 的并发控制

12.1 Spanner 背景

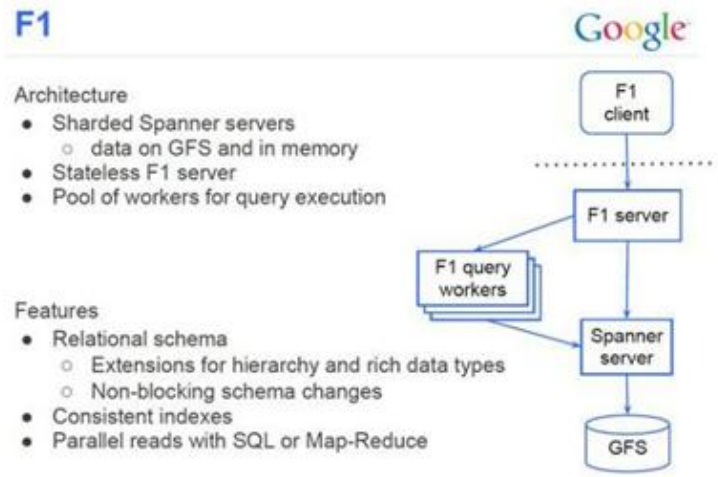

图 12-1 Spanner 在 Google 公司中的定位

要想深刻理解 Spanner 的原理，必须要首先了解 Spanner 在 Google 的定位。从图 12-1 可以看到。Spanner 位于 F1 和 GFS 之间，承上启下。所以，这里先简要介绍 F1 和 GFS。

12.1.1 F1

和众多互联网公司一样，在早期 Google 大量使用了 MySQL。MySQL 是单机的，可以用 Master-Slave 来容错，通过分区来实现扩展。但是，需要大量的手工运维工作，有很多的限制。因此，Google 开发了一个可容错、可扩展的 RDBMS——F1。和一般的分布式数据库不同，F1 对于 RDBMS 应有的功能毫不妥协。起初 F1 是基于 MySQL 的，不过会逐渐迁移到 Spanner。

F1 有如下特点：

- 7×24 高可用，哪怕某一个数据中心停止运转，仍然可用；
- 可以同时提供强一致性和弱一致；
- 可扩展；
- 支持 SQL；
- 事务提交延迟 50-100ms，读延迟 5-10ms，高吞吐。

众所周知，Google BigTable 是重要的 NoSQL 产品，提供很好的扩展性，开源世界有 HBase 与之对应。为什么 Google 还需要 F1，而不是都使用 BigTable 呢？这是因为，BigTable 提供的“最终一致性”，无法满足一些需要强事务一致性的应用的需求。同时，BigTable 还是 NoSQL，而大量的应用场景需要关系模型（现在大量的互联网企业都使用 MySQL 而不愿意使用 HBase）。因此，Google 才设计了可扩展数据库 F1，支持关系模型，而 Spanner 就是 F1 的至关重要的底层存储技术。

12.1.2 Colossus（GFS II）

Colossus 也是一个不得不提起的技术，它是第二代 GFS，对应于开源世界的新 HDFS。

GFS 是著名的分布式文件系统。第一代 GFS 是为批处理而设计的，对于大文件很友好，吞吐量很大，但是延迟较高。所以，使用它的系统不得不对 GFS 做各种优化，才能获得良好的性能。那为什么 Google 没有考虑到这些问题，设计出更完美的 GFS 呢？这是因为，那个时候是 2001 年，Hadoop 出生是在 2007 年。如果 Hadoop 是世界领先水平的话，GFS 比世界领先水平还领先了 6 年。同样地，Spanner 面世时间大概是 2009 年，等到我们看到论文的时候（注：Google Spanner 英文论文于 2012 年 9 月发布），估计 Spanner 在 Google 已经很完善了，同时可以预见到，Google 内部应该已经有更先进的替代技术在酝酿了。最早在 2015

年才会出现 Spanner 和 F1 的开源产品。

Colossus 是第二代 GFS。Colossus 是 Google 重要的基础设施，因为，它可以满足主流应用对 GFS 的要求。Colossus 的重要改进有：

- 优雅 Master 容错处理 (不再有 2s 的停止服务时间)；
- Chunk 大小只有 1MB (对小文件很友好)；
- Master 可以存储更多的 Metadata(当 Chunk 从 64MB 变为 1MB 后，Metadata 会扩大 64 倍，但是 Google 也解决了)。
- Colossus 可以自动分区 Metadata。使用 Reed-Solomon 算法来复制，可以将原先的 3 份减小到 1.5 份，提高写的性能，降低延迟。

12.2与 BigTable、Megastore 的对比

Spanner 主要致力于跨数据中心的数据复制上，同时也能提供数据库功能。在 Google 类似的系统还有 BigTable 和 Megastore。和这两者相比，Spanner 又有什么优势呢？

BigTable 在 Google 得到了广泛的使用，但是，它不能提供较为复杂的 Schema，也无法提供在跨数据中心环境下的强一致性。Megastore 有类似于 RDBMS 的数据模型，同时也支持同步复制，但是，它的吞吐量太差，不能适应应用要求。Spanner 不再是类似 BigTable 的版本化 key-value 存储，而是一个“临时多版本”的数据库。所谓的“临时多版本”是指，数据是存储在一个版本化的关系表里面，存储的数据会根据其提交的时间打上时间戳，应用可以访问到较老的版本，另外，老的版本也会被垃圾回收掉。

Google 官方认为 Spanner 是下一代 BigTable，也是 Megastore 的继任者。

12.3 Spanner 的功能

从高层看，Spanner 是通过 Paxos 状态机将分区好的数据分布在全球的。数据复制是全球化的，用户可以指定数据复制的份数和存储的地点。 Spanner 可以在集群或者数据发生变化的时候将数据迁移到合适的地点，做负载均衡。用户可以指定将数据分布在多个数据中心，不过更多的数据中心将造成更多的延迟。用户需要在可靠性和延迟之间做权衡，一般来说，复制 1、2 个数据中心足以保证可靠性。

作为一个全球化分布式系统，Spanner 提供一些有趣的特性，主要包括：

（1）应用可以细粒度地指定数据分布的位置，精确地指定数据离用户有多远，可以有

效地控制读延迟(读延迟取决于最近的拷贝)；可以指定数据拷贝之间有多远，可以控制写的延迟(写延迟取决于最远的拷贝)；还可以指定数据的复制份数，控制数据的可靠性和读性能(多写几份数据，可以抵御更大的事故)；

（2）Spanner 还有两个一般分布式数据库不具备的特性：读写的外部一致性和基于时间戳的全局的读一致。这两个特性可以让 Spanner 支持一致的备份，一致的 MapReduce，还有原子的 Schema 修改。

这些特性都得益于 Spanner 有一个全球时间同步机制，可以在数据提交的时候给出一个时间戳。因为时间是序列化的，所以才有外部一致性。这个很容易理解，如果有两个提交，一个在时间 T1，一个在时间 T2，那么，更晚的时间戳那个提交是正确的。

这个全球时间同步机制是由一个具有 GPS 和原子钟的 TrueTime API 提供的。这个 TrueTime API 能够将不同数据中心的时间偏差缩短到 10ms 以内。这个 API 可以提供一个精确的时间，同时给出误差范围。Google 已经有了一个 TrueTime API 的实现。这个 TrueTimeAPI 非常有意义，如果能单独开源实现这部分机制的话，很多数据库（如 MongoDB）都可以从中受益。

12.4 体系结构

Spanner 由于是全球化的，所以有两个其它分布式数据库所没有的概念：

- **Universe**：一个 Spanner 部署实例，称为一个 Universe。目前全世界有 3 个，一个开发，一个测试，一个线上。因为一个 Universe 就能覆盖全球，因此，不需要多个。
- **Zone**：每个 Zone 相当于一个数据中心，一个 Zone 内部物理上必须在一起。而一个数据中心可能有多个 Zone。可以在运行时添加或移除 Zone。一个 Zone 可以理解为一个 BigTable 部署实例。

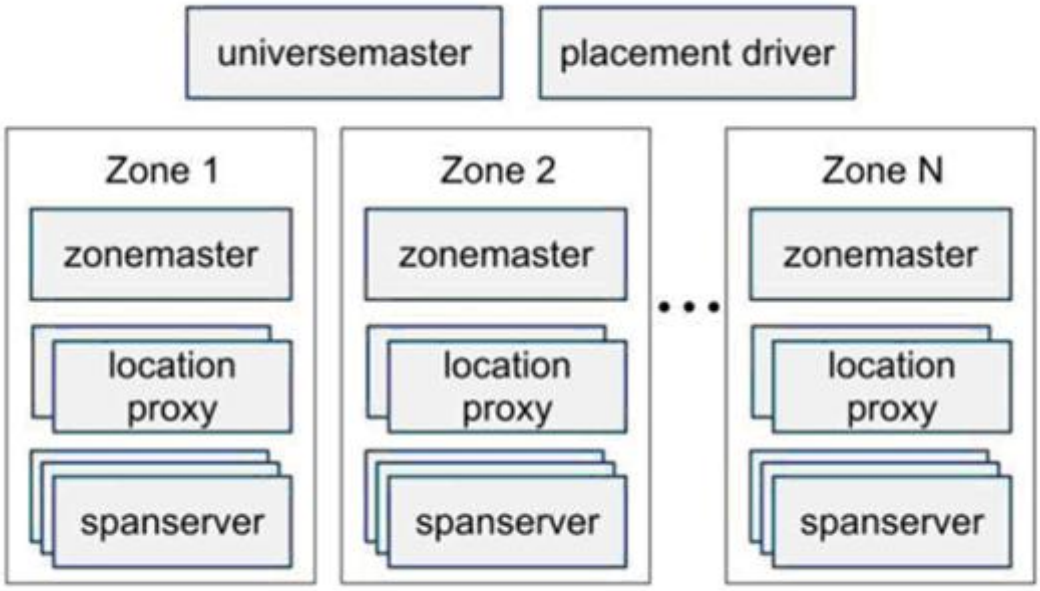

图 12-2 Spanner 体系结构

图 12-2 显示了在一个 Spanner 的 universe 中的服务器架构。一个 zone 包括一个 zonemaster 和一百至几千个 spanserver。Zonemaster 把数据分配给 spanserver，spanserver 把数据提供给客户端。客户端使用每个 zone 上面的 location proxy 来定位可以为自己提供数据的 spanserver。Universe master 和 placement driver，当前都只有一个。Universe master 主要是一个控制台，它显示了关于 zone 的各种状态信息，可以用于相互之间的调试。Placement driver 会周期性地与 spanserver 进行交互，来发现那些需要被转移的数据，或者是为了满足新的副本约束条件，或者是为了进行负载均衡。

简单总结一下，一个 Spanner 主要包括以下组件：

- Universemaster：监控一个 universe 里 zone 级别的状态信息；
- Placement driver：提供跨区数据迁移时的管理功能；
- Zonemaster：相当于 BigTable 的 Master，管理 Spanserver 上的数据；
- Location proxy：存储数据的 Location 信息，客户端要先访问它才能知道数据在哪个 Spanserver 上；
- Spanserver：相当于 BigTable 的 ThunkServer，用于存储数据。

12.5 Spanserver

本节详细介绍 Spanserver 的设计实现。Spanserver 的设计和 BigTable 非常地相似。

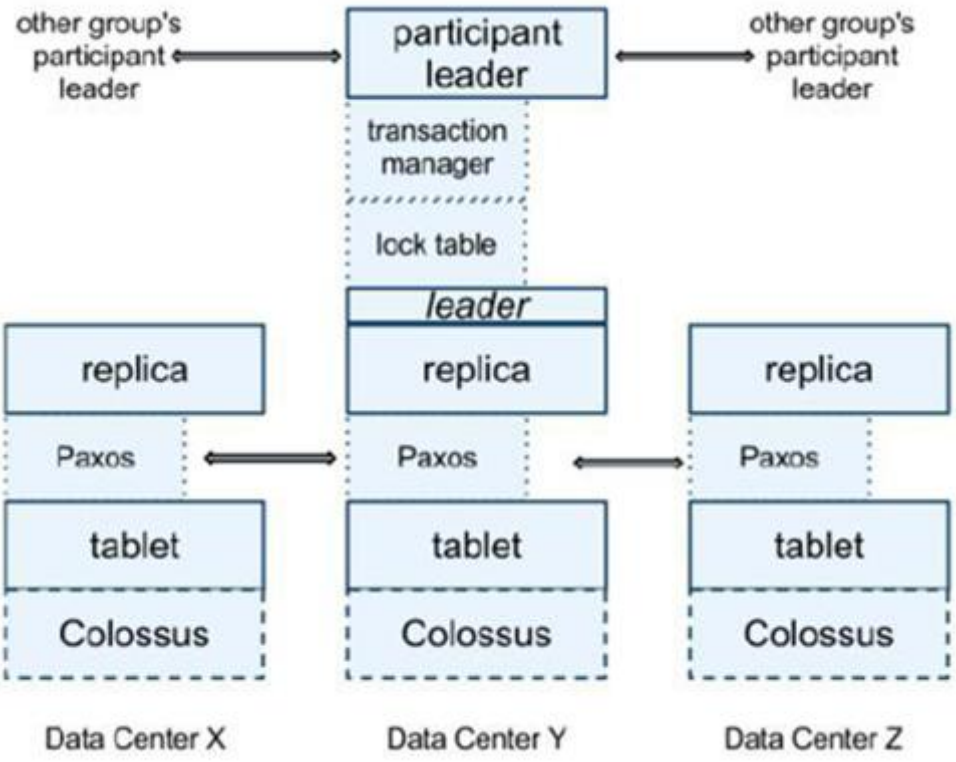

图 12-3 Spanserver 软件栈

如图 12-3 所示，从下往上看，每个数据中心会运行一套 Colossus (GFS II) ，每个机器有 100-1000 个 tablet。Tablet 在概念上将相当于数据库一张表里的一些行，物理上是数据文件。打个比方，一张 1000 行的表，有 10 个 tablet，第 1-100 行是一个 tablet，第 101-200 是一个 tablet。一个 tablet 就类似于 BigTable 中的 tablet，也实现了下面的映射：

(key:string, timestamp:int64)->string。

与 BigTable 不同的是，Spanner 会把时间戳分配给数据，这种非常重要的方式，使得 Spanner 更像一个多版本数据库，而不是一个键值存储。一个 tablet 的状态是存储在类似于 B-树的文件集合和写前(write-ahead)日志中的，所有这些都会被保存到一个分布式的文件系统中，这个分布式文件系统被称为 Colossus，它继承自 Google File System。

在 Spanner 中，每个 Tablet 上会有一个 Paxos 状态机（Paxos 是一个分布式一致性协议），Table 的元数据和日志都存储在上面。Paxos 会选出一个副本（replica）做 leader，这个 leader 的寿命默认是 10s，10s 后重选。Leader 就相当于复制数据的 master，其他副本的数据都是从它那里复制的。读请求可以走任意的部分，但是，写请求只有去找 leader。这些副本统称为一个 paxos group。

每个 leader replica 在 spanserver 上会实现一个锁表来管理并发。锁表记录了两阶段提交需要的锁信息。但是，不论是在 Spanner 还是在 BigTable 上，当遇到冲突的时候，长事务的

性能会表现得很差。所以，有一些操作（如事务读）可以走锁表，其它的操作可以绕开锁表。

每个 leader replica 的 spanserver 上还有一个事务管理器。如果事务在一个 paxos group 里面，可以绕过事务管理器。但是，一旦事务跨多个 paxos group，就需要事务管理器来协调。其中一个事务管理器被选为 leader，其它的事务管理器是 slave，听从 leader 的指挥。这样可以保证事务性。

12.6 Directory

之所以 Spanner 比 BigTable 有更强的扩展性，主要原因在于，Spanner 还有一层抽象的概念——directory。directory 是一些 key-value 的集合，一个 directory 里面的 key 有一样的前缀，更妥当的叫法是 bucketing。 Directory 是应用控制数据位置的最小单元，可以通过谨慎地选择 key 的前缀来控制。

Directory 作为数据放置的最小单元，可以在 paxos group 里面移来移去。Spanner 移动一个 directory 一般出于如下几个原因：

- 一个 paxos group 的负载太大，需要切分；
- 将数据移动到距离访问者更近的地方；
- 将经常同时访问的 directory 放到一个 paxos group 里面。

Directory 可以在不影响 client 的前提下，在后台移动。移动一个 50MB 的 directory 大概需要几秒钟时间。

那么 directory 和 tablet 又是什么关系呢？可以这么理解，Directory 是一个抽象的概念，管理数据的单元；而 tablet 是物理的东西，数据文件。由于一个 Paxos group 可能会有多个 directory，所以，Spanner 的 tablet 实现和 BigTable 的 tablet 实现有些不同。BigTable 的 tablet 是单个顺序文件。而 Spanner 的 tablet 可以理解为是一些基于行的分区的容器。这样就可以将一些经常同时访问的 directory 放在一个 tablet 里面，而不用太在意顺序关系。

在 paxos group 之间移动 directory 是后台任务，这个操作还被用来移动副本。移动操作在设计的时候并没有采用事务来实现，否则，会造成大量的读写阻塞（block）。操作的时候，需要先将实际数据移动到指定位置，然后再用一个原子的操作更新元数据，这样就完成了整个移动过程。

Directory 还是记录地理位置的最小单元。数据的地理位置是由应用决定的，配置的时候需要指定复制数目和类型，还有地理的位置(比如上海复制 2 份；南京复制 1 份) 。

12.7 数据模型

Spanner 的数据模型来自于 Google 内部的实践。在设计之初，Spanner 就决心有以下的特性：

- 支持类似关系数据库的 schema；
- Query 语句；
- 支持广义上的事务。

为何会这样决定呢？在 Google 内部还有一个 Megastore，尽管要忍受性能不够的折磨，但是在 Google 还有 300 多个应用在用它，因为，Megastore 支持一个类似关系数据库的 schema，而且支持同步复制 (BigTable 只支持最终一致的复制) 。使用 Megastore 的应用包括大名鼎鼎的 Gmail、Picasa、Calendar、Android Market 和 AppEngine 等等。 而必须对 Query 语句的支持，则来自于广受欢迎的 Dremel，它可以支持在 2-3 秒内实现对 PB 级别数据的查询。最后，对事务的支持是必不可少的了，BigTable 在 Google 内部被抱怨的最多的就是其只能支持行级事务，再大粒度的事务就无能为力了。Spanner 的开发者认为，过度使用事务造成的性能下降的恶果，应该由应用的开发者来承担；应用开发者在使用事务的时候，必须考虑到性能问题；而数据库必须提供事务机制，而不是因为性能问题就干脆不提供事务支持。

Spanner 的数据模型是建立在 directory 和 key-value 模型的抽象之上的。一个应用可以在一个 universe 中建立一个或多个 database，在每个 database 中建立任意的 table。Table 看起来就像关系型数据库的表，有行，有列，还有版本。Query 语句看起来是多了一些扩展的 SQL 语句。

Spanner 的数据模型也不是纯正的关系模型，每一行都必须有一列或多列组件，看起来还是 Key-value，主键组成 Key，其他的列是 Value。但是，这样的设计对应用而言也是很有裨益的，应用可以通过主键来定位到某一行。

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

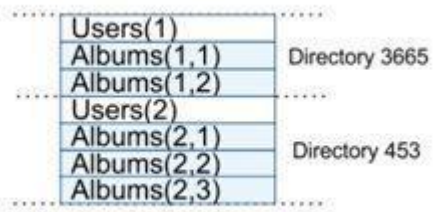

图 12-4 Spanner 模式实例

图 12-4 是一个 Spanner 模式的例子。对于一个典型的相册应用，需要存储其用户和相册，可以用上面的两个 SQL 来创建表。Spanner 的表是层次化的，最顶层的表是 directory table，其它的表在创建的时候，可以用 interleave in parent 来声明层次关系。这样的结构在实现的时候，Spanner 可以将嵌套的数据放在一起，这样在分区的时候性能会提升很多。否则，Spanner 无法获知最重要的表之间的关系。

12.8 TrueTime

Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(*t*)	true if *t* has definitely passed
TT.before(*t*)	true if *t* has definitely not arrived

表 12-1 TrueTime API

TrueTime API（如表 12-1 所示）是一个非常有创意的东西，可以同步全球的时间。TT.now()可以获得一个绝对时间 TTinterval，这个值和 UnixTime 是相同的，同时，还能够得到一个误差 e。TT.after(t)和 TT.before(t)是基于 TT.now()实现的。

在底层，TrueTime 使用的时间是 GPS 和原子钟。TrueTime 使用两种类型的时间，是因为它们有不同的失败模式。GPS 参考时间的弱点是天线和接收器失效、局部电磁干扰和相关失败（比如设计上的缺陷导致无法正确处理闰秒和电子欺骗），以及 GPS 系统运行中断。原子钟也会失效，不过失效的方式和 GPS 无关，不同原子钟之间的失效也不存在彼此相关

性。由于存在频率误差，在经过很长的时间以后，原子钟也会产生明显误差。

TrueTime 是由每个数据中心上面的许多 time master 机器和每台机器上的一个 timeslave daemon 来共同实现的。大多数 master 都有具备专用天线的 GPS 接收器，这些 master 在物理上是相互隔离的，这样可以减少天线失效、电磁干扰和电子欺骗的影响。剩余的 master（我们称为 Armageddon master）则配备了原子钟。一个原子钟并不是很昂贵：一个 Armageddon master 的花费和一个 GPS master 的花费是同一个数量级的。所有 master 的时间参考值都会进行彼此校对。每个 master 也会交叉检查时间参考值和本地时间的比值，如果二者差别太大，就会把自己驱逐出去。在同步期间，Armageddon master 会表现出一个逐渐增加的时间不确定性，这是由保守应用的最差时钟漂移引起的。GPS master 表现出的时间不确定性几乎接近于 0。

每个 daemon 会从许多 master 中收集投票，获得时间参考值，从而减少误差。被选中的 master 中，有些 master 是 GPS master，是从附近的数据中心获得的，剩余的 GPS master 是从远处的数据中心获得的；还有一些是 Armageddon master。Daemon 会使用一个 Marzullo 算法的变种，来探测和拒绝欺骗，并且把本地时钟同步到非撒谎 master 的时间参考值。为了免受较差的本地时钟的影响，Spanner 会根据组件规范和运行环境确定一个界限，如果机器的本地时钟误差频繁超出这个界限，这个机器就会被驱逐出去。

在同步期间，一个 daemon 会表现出逐渐增加的时间不确定性。ε 是从保守应用的最差时钟漂移中得到的。ε 也取决于 time master 的不确定性，以及与 time master 之间的通讯延迟。在 Google 的线上应用环境中，ε 通常是一个关于时间的锯齿形函数。在每个投票间隔中，ε 会在 1 到 7ms 之间变化。因此，在大多数情况下，ε 的平均值是 4ms。Daemon 的投票间隔，在当前是 30 秒，当前使用的时钟漂移比率是 200 微秒/秒，二者一起意味着 0 到 6ms 的锯齿形边界。剩余的 1ms 主要来自到 time master 的通讯延迟。在失败的时候，超过这个锯齿形边界也是有可能的。例如，偶尔的 time master 不确定性，可能会引起整个数据中心范围内的ε 值的增加。类似地，过载的机器或者网络连接，都会导致ε 值偶尔地局部增大。

12.9 Spanner 的并发控制

Spanner 使用 TrueTime 来控制并发，实现外部一致性，主要支持以下几种事务：

- 读写事务；
- 只读事务；

- 快照读，客户端提供时间戳；
- 快照读，客户端提供时间范围。

例如，一个读写事务发生在时间 t，那么，在全世界任何一个地方，指定 t 快照读都可以读到写入的值。

表 12-2 Spanner 支持的事务类型

Operation	Concurrency Control	Replica Required
Read-Write Transaction	pessimistic	leader
Read-Only Transaction	lock-free	leader for timestamp; any for read
Snapshot Read, client-provided timestamp	lock-free	any
Snapshot Read, client-provided bound	lock-free	any

表 12-2 是 Spanner 现在支持的事务。单独的写操作都被实现为读写事务；单独的非快照被实现为只读事务。事务总有失败的时候，如果失败，对于这两种操作会自己重试，无需应用自己实现重试循环。

时间戳的设计大大提高了只读事务的性能。事务开始的时候，要声明这个事务里没有写操作。只读事务可不是一个简单的没有写操作的读写事务，它会用一个系统时间戳去读，所以，对于同时的其他的写操作是没有阻塞的。而且，只读事务可以在任意一台已经更新过的副本上面读。

对于快照读操作，可以读取以前的数据，需要客户端指定一个时间戳或者一个时间范围。Spanner 会找到一个已经充分更新好的副本上读取。

还有一个有趣的特性的是，对于只读事务，如果执行到一半，该副本出现了错误，客户端没有必要在本地缓存刚刚读过的数据，因为，数据是根据时间戳读取的，只要再用刚刚的时间戳去读取，就可以获得一样的结果。

- **读写事务**

正如 BigTable 一样，Spanner 的事务是会将所有的写操作先缓存起来，在 Commit 的时候一次提交。这样的话，就读不出在同一个事务中写的数据了。不过这没有关系，因为 Spanner 的数据都是有版本的。

Spanner 在读写事务中使用 wound-wait 算法来避免死锁。当客户端发起一个读写事务的时候，首先是读操作，它先找到相关数据的 leader replica，然后加上读锁，读取最近的数据。在客户端事务存活的时候会不断地向 leader 发心跳，防止超时。当客户端完成了所有的读操作，并且缓存了所有的写操作，就开始了两阶段提交。客户端选择一个 coordinator，并给每一个 leader 发送 coordinator 的 id 和缓存的写数据。

leader 首先会上一个写锁，它要找一个比现有事务晚的时间戳。通过 Paxos 记录，每一个相关的 leader 都要给 coordinator 发送它自己准备的那个时间戳。

Coordinator 一开始也会上个写锁，当大家发送时间戳给它之后，它就选择一个提交时间戳。这个提交的时间戳，必须比刚刚的所有时间戳晚，而且还要比 TT.now()+误差时间还要晚。这个 Coordinator 将这个信息记录到 Paxos。

在让副本写入数据生效之前，coordinator 还要再等一会儿，需要等两倍时间误差。这段时间也刚好让 Paxos 来同步。因为等待之后，在任意机器上发起的下一个事务的开始时间，都不会比这个事务的结束时间早了。然后，coordinator 将提交时间戳发送给客户端还有其它的副本，它们记录日志，写入生效，释放锁。

- **只读事务**

对于只读事务，Spanner 首先要指定一个读事务时间戳，还需要了解在这个读操作中，需要访问的所有的读的 Key。Spanner 可以自动确定 Key 的范围。

如果 Key 的范围在一个 Paxos group 内。客户端可以发起一个只读请求给 group leader。leader 选一个时间戳，这个时间戳要比上一个事务的结束时间要大，然后读取相应的数据。这个事务可以满足外部一致性，读出的结果是最后一次写的结果，并且不会有不一致的数据。

如果 Key 的范围在多个 Paxos group 内，就相对复杂一些。其中一个比较复杂的例子是，可以遍历所有的 group leaders，寻找最近的事务发生的时间并读取。客户端只要时间戳在 TT.now().latest 之后就可以满足要求了。

本章小结

本章首先介绍了 Spanner 的诞生背景，并把 Spanner 与 BigTable、Megastore 进行了对比；接下来简单介绍了 Spanner 的功能和体系结构；然后，介绍了 Spanner 的具体设计方法，包括 Spanserver、Directory、数据模型、TrueTime 等，最后，介绍了 Spanner 的并发控制机制。

参考文献

[1] 林子雨翻译. Google Spanner(中文版). http://dblab.xmu.edu.cn/node/230

[2] 颜开. 全球级的分布式数据库 Google Spanner 原理.

http://www.oschina.net/question/12_70811

第 13 章 Google Dremel

Dremel 是一种可扩展的、交互式的实时查询系统，用于只读嵌套（nested）数据的分析。通过结合多级树状执行过程和列式数据结构，它能做到几秒内完成对万亿张表的聚合查询。系统可以扩展到成千上万的 CPU 上，满足 Google 上万用户操作 PB 级的数据，可以在 2 到 3 秒内完成 PB 级别数据的查询。在本章中，我们将描述 Dremel 的架构和实现，解释它为何是 MapReduce 计算的有力补充。此外，我们也描述了一种新的针对嵌套记录的列存储形式。

本章介绍 Dremel 的相关知识，内容要点如下：

- Dremel 概述
- Dremel 的数据模型
- 嵌套列式存储
- 查询语言
- 查询的执行

13.1Dremel 概述

13.1.1 大规模数据分析

大规模分析型数据处理，在互联网公司乃至整个行业中都已经越来越广泛。目前已经可以用廉价的存储来收集和保存海量的企业核心业务数据，但是，更重要的是，必须懂得如何让分析师和工程师便捷地利用这些数据。在数据探测、监控、在线用户支持、快速原型设计、数据管道调试以及其他任务中，交互的响应时间是一个至关重要的系统设计考虑因素。

执行大规模交互式数据分析对并行计算能力要求很高，例如，假设使用普通的硬盘，如果希望在 1 秒内读取 1TB 的压缩数据，那么就需要成千上万块硬盘。类似地，CPU 密集的查询操作也需要运行在成千上万个核上，并在数秒内完成。在 Google 公司里，大量的并行计算是使用普通 PC 组成的共享集群完成的。一个集群通常会部署大量“共享资源、产生不同负载、需要不同硬件参数”的分布式应用。对于一个分布式应用而言，某个工作任务可能会比其他任务花费更多的时间，或者可能由于故障，或者被集群管理系统停止而永远不能完成。因此，处理好异常和故障是实现快速执行和容错的重要因素。

互联网和科学计算中的数据经常是没有关联的，因此，在这些领域，一个灵活的数据模

型是十分必要的。在编程语言中使用的数据结构、分布式系统之间交换的消息、结构化文档等等，都可以用嵌套方式来很自然地描述。嵌套数据模型已经成为 Google 处理大部分结构化数据的基础，据报道，其他互联网公司也在使用这种嵌套数据模型。

13.1.2 Dremel 的特点

随着 Hadoop 的流行，大规模的数据分析系统已经越来越普及。但是，Hadoop 比较适合用于大规模数据的批量处理，而对于实时的交互式处理就有点显得力不从心，比如，Hadoop 通常无法做到让用户在 2 到 3 秒内迅速完成 PB 级别数据的查询。因此，数据分析师需要一个能将数据“玩转”的交互式系统，这样就可以非常方便快捷地浏览数据以及建立分析模型。Google 公司设计的 Dremel 就是一个能够满足这种实时交互式处理的系统，它具有以下几个主要的特点：

（1）Dremel 是一个大规模、稳定的系统。在一个 PB 级别的数据集上面，将任务缩短到秒级，无疑需要大量的并发。Google 一向是用廉价机器办大事的好手，但是，机器越多，出问题概率越大。如此大的集群规模，需要有足够的容错考虑，保证整个分析的速度不被集群中的个别慢(坏)节点所影响。

（2）Dremel 是 MapReduce 交互式查询能力不足的补充。和 MapReduce 一样，Dremel 也需要和数据运行在一起，将计算移动到数据上面，所以，它需要 GFS 这样的文件系统作为存储层。在设计之初，Dremel 并非是 MapReduce 的替代品，它只是可以执行非常快的分析，在使用的时候，常常用它来处理 MapReduce 的结果集或者用来建立分析原型。

（3）Dremel 的数据模型是嵌套(nested)的。互联网数据常常是非关系型的，这就要求 Dremel 必须有一个灵活的数据模型，这个数据模型对于获得高性能的交互式查询而言至关重要。因此，Dremel 采用了嵌套(nested)数据模型，有点类似于 Json。嵌套数据模型相对于关系模型而言具有明显的优势。对于传统的关系模型而言，不可避免地存在大量 Join 操作，因此，在处理如此大规模的数据的时候，往往是有心无力的。而嵌套数据模型却可以在 PB 级别数据上一展身手。

（4） Dremel 中的数据是用列式存储的。当采用列式存储时，在分析的时候就可以只扫描需要的那部分数据，从而大大减少 CPU 和磁盘的访问量。同时，列式存储是压缩友好的，可以实现更高的压缩率，使得 CPU 和磁盘发挥最大的效能。对于关系型数据而言，在如何使用列式存储方面，我们都已经很有经验。但是，对于嵌套(nested)结构，Dremel 也通

过巧妙的设计来实现列式存储，这是非常值得我们学习的。

（5） Dremel 结合了 Web 搜索 和并行 DBMS 的技术。首先，它借鉴了 Web 搜索中的“查询树”的概念，将一个相对巨大复杂的查询分割成较小、较简单的查询。大事化小，小事化了，能并发地在大量节点上跑。其次，和并行 DBMS 类似，Dremel 可以提供了一个类似 SQL 的接口，就像 Hive 和 Pig 那样。

Dremel 自从 2006 年开始就已经投入开发了，并且在 Google 公司已经有了几千用户。多种多样的 Dremel 实例被部署在 Google 公司里，每个实例拥有着数十至数千个节点。使用 Dremel 系统的例子包括：

- 分析网络文档；
- 追踪 Android 市场应用程序的安装数据；
- Google 产品的崩溃报告分析；
- Google Books 的 OCR 结果 ；
- 垃圾邮件分析；
- Google Maps 里地图部件调试；
- 管理中的 Bigtable 实例的 Tablet 迁移；
- Google 分布式构建系统中的测试结果分析；
- 数万个硬盘的磁盘 IO 统计信息；
- Google 数据中心上运行的任务的资源监控；
- Google 代码库的符号和依赖关系分析。

13.1.3 Dremel 的应用场景

为了说明交互式查询处理如何融入更广泛的数据管理领域，这里设计了一个应用场景。假想有一名 Google 的工程师 Alice，想从大量网页中提取新的信息。她运行 MapReduce 任务从输入数据中跑出数十亿条包括所有信息的记录，村粗到分布式文件系统中。为了分析实验结果，她通过 Dremel 执行了几条交互式命令：

```
DEFINE TABLE t AS /path/to/data/*
SELECT TOP(signal1, 100), COUNT(*) FROM t
```

这些命令在几秒内就执行完了，之后她又做了其它一些查询来验证她的算法是否工作正常。她发现了 signal1 中有一个不正确的地方，为了更深入地考察其中的问题，她写了个

FlumeJava 程序，在之前输出的数据集上进行一些更复杂的分析计算。这一步解决后，她又创建了一个管道来不停地处理新进来的数据。此外，她还写了些 SQL 查询来聚合管道各个维度的结果输出，并把它加到了交互式的操作界面上。最后，她在一个目录里声明了她的新数据集，这样其他工程师可以找到并快速查询。

上述案例要求在查询处理器和其他数据管理工具之间互相协作。第一个组成部分是一个公用的存储层。GFS（Google File System）是公司中广泛使用的分布式存储层。GFS 使用冗余复制来保护数据不受硬盘故障影响，即使出现异常也能达到快速响应时间。对数据管理来说，一个高性能的存储层是非常重要的，它允许访问数据时不消耗太多时间在加载阶段。这个要求也导致数据库在分析型数据处理中不常被使用，因为，在 DBMS 加载数据以及执行单一查询前，可能要运行数个 MapReduce 分析任务。使用 GFS 的另外一个好处是，在文件系统中能使用标准工具便捷地操作数据，比如，迁移到另外的集群，改变访问权限，或者基于文件名定义一个数据子集的分析。

构建互相协作的数据管理组件的第二个要素是一个共享的存储格式。列式存储已经被证明适用于扁平的关系型数据，但是，使它适用于 Google 则需要转换到一个嵌套数据模型。图 13-1 展示了对嵌套数据进行列式存储的主要思想：一个嵌套字段比如 A.B.C，它的所有值被连续存储。因此，A.B.C 被读取时，不需读取 A.E、A.B.D 等等。

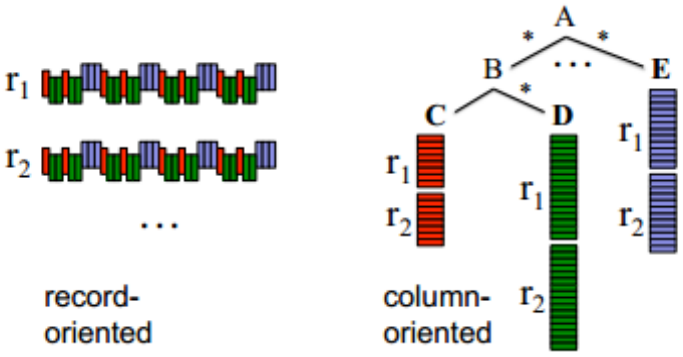

图 13-1 嵌套数据的行式存储和列式存储

13.2Dremel 数据模型

本节我们介绍 Dremel 的数据模型以及一些后续将会用到的术语。这个数据模型是基于强类型嵌套记录的，它的抽象语法是：

π = dom | <A1 : π[*|?],…,An : π[*|?]>

π 是一个原子类型或者记录类型。在 dom 中原子类型包含整型、浮点数、字符串等等。

记录则由一或多个字段组成。在记录中字段 i 标记为 Ai，以及拥有一个可选的多样性的标签。另外需要注意的是字段的类型，每个字段都属于某种类型，比如，required 表示有且仅有一个值；optional 表示“可选”，有 0 到 1 个值；repeated（*），表示“重复”，有 0 到 N 个值。其中，repeated 和 optional 类型是非常重要的，从它们身上抽象出一些重要的概念，以便用最少的代价来无损地描述出原始的数据。

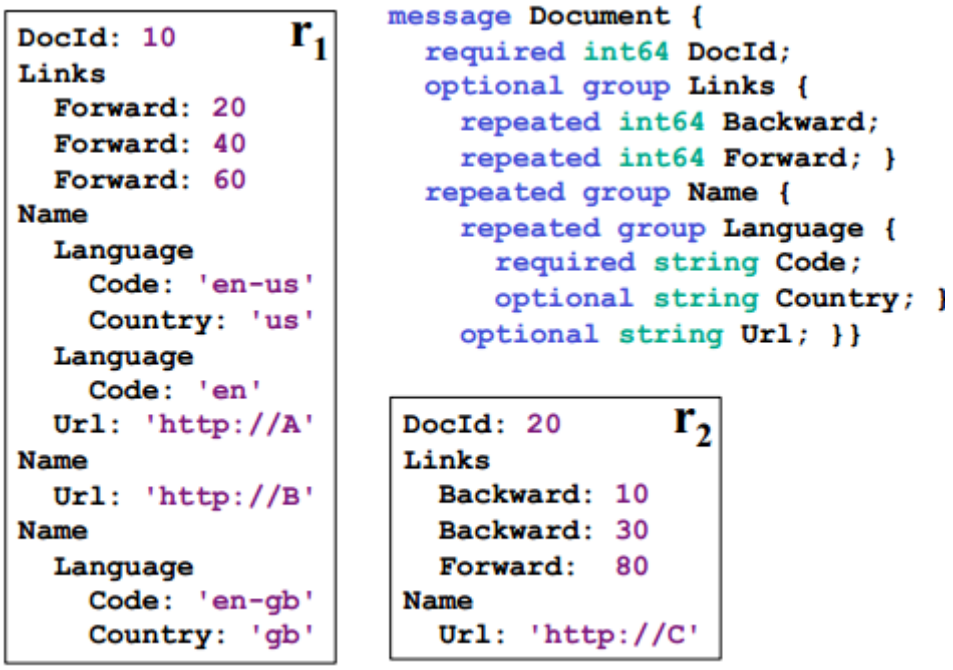

图 13-2 两个简单的嵌套记录和它们的 schema

图 13-2 描述了两个简单的嵌套记录和它们的 schema，用来表示一个网页。该 schema 定义使用了上面介绍的语法。一个网页文档拥有必需的整型 DocId 属性和可选的 Links 属性，以及包含在列表中的 Forward 和 Backward，列表中每一项代表其他网页的 DocId。一个网页文档可以有多个 Name，代表该网页所被引用的不同 URL。Name 包含一系列 Code 和 Country（可选）的组合。图 13-2 也展现了遵循上述 schema 的两个示例记录，即 r1 和 r2。记录的结构通过缩进体现出来。后续内容中，我们将使用这些记录的例子来解释相应的算法。Schema 中定义的字段形成了一个树状结构。一个嵌套字段的完整路径，是通过点号来表示，如 Name.Language.Code。

嵌套数据模型为 Google 的序列化、结构化数据奠定了一个平台无关的可扩展机制。代码生成工具生成具体编程语言（如 C++、Java）的代码。跨语言的兼容性是通过对记录的标准二进制化表示来保证的，记录中的字段及相应的值被序列化后进行传输。通过这种方式，Java 写的 MapReduce 程序可以处理另外一个 C++生成的数据源中的记录。因此，如果记录

采用列式存储，快速的记录重建（即从列式存储中组装出原来的记录），对于 MapReduce 和其它数据处理工具都非常重要。

13.3嵌套列式存储

如图 13-1 所示，我们的目标是连续地存储一个给定的字段的所有值来改善查询效率。在本节中，我们阐述了要解决的问题：一个列式格式记录的无损表示、分割记录为列式存储、高效的记录装配。

13.3.1 重复深度、定义深度

让我们重新审视一下图 13-1 右边的列式存储结构，这是 Dremel 的目标，它就是要将图 13-2 中 Document 那种嵌套结构转变为列式存储结构。实现这个目标的方式多种多样，Google 信心满满地推出了它设计的最优化、最节省成本、效率最高的方法，并且引出了两个全新的概念，即重复深度和定义深度。因为 Dremel 会将记录肢解、再按字段各自集中存储，此举难免会导致数据失真，比如图 13-2 中，我们把 r1 和 r2 的 URL 列值放在一起得到[“http://A”,"http://B","http://C"]，那么，我们怎么知道它们各自属于哪条记录、属于记录中的哪个 Name 呢？这里提出的两个概念——重复深度和定义深度，其实就是为了解决这种失真问题，从而实现无损表达。

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

图 13-3 演示重复深度和定义深度的实例

- **重复深度（Repetition Level）**

注意在图 13-2 中的 Code 字段，可以看到它在 r1 中总共出现了 3 次。‘en-us’、‘en’在第一个 Name 中，而‘en-gb’在第三个 Name 中。为了消除这种字段值的含糊性，我们对字段值附加了“重复深度”，它可以告诉我们，在路径中的哪个重复字段重复出现了，依此来确定此值的位置。比如，在 Name.Language.Code 这个路径中，包含两个重复字段，Name 和 Language，因此，Code 字段的重复深度范围为 0 到 2，其中，0 意味着一个新记录的开始。现在让我们从上至下扫描纪录 r1，来看看重复深度的具体含义。当我们遇到‘en-us’，我们没看到任何重复的字段，也就是说，重复深度是 0。当我们遇到‘en’，字段 Language 重复了，Language 在 Name.Language.Code 中排在第 2 个级别，所以重复深度是 2。最后，当我们遇到‘en-gb’，Name 重复了（Name 后 Language 只出现过一次，因此要注意，这里的 Language 是没有重复的），Name 在 Name.Language.Code 中排在第 1 个级别，所以重复深度是 1。因此，从上至下扫描纪录 r1 以后，可以得到 r1 中 Code 的三个值（‘en-us’、 ‘en’和‘en-gb’）的重复深度分别是 0、2、1。

这里要注意，r1 中的第二个 Name 没有包含任何 Code 值。在重构记录时，为了确定‘en-gb’出现在第三个 Name 而不是第二个 Name 中，我们需要额外添加一个 NULL 值在‘en’和‘en-gb’之间（如图 13-3 所示）。在 Language 字段中，Code 字段是必需的值，也就是说，只要 Language 字段出现，就一定会有 Code 字段出现并且有值，所以，Code 字段值的缺失意味着 Language 也没有定义。因此，在 r1 中的第二个 Name 中，没有 Code 字段值，就意味着没有 Language 字段值。一般来说，确定一个路径中（比如 Name.Language.Code 这条路径）有哪些字段被明确定义，需要一些额外的信息，这就需要引入“定义深度”的概念。

- **定义深度（Definition Level）**

定义深度从某种意义上来说是服务于重复深度的，因为，在重新装配记录时，只有在“仅靠重复深度无法确定字段值的位置”时（一般是字段值为 NULL 的情形），才需要去参考定义深度来确定字段值的位置。在 Dremel 系统中，所有列都是先存储来自 r1 的内容，后存储来自 r2 的内容，也就是说，对于所有的列而言，记录存储的顺序是一致的。这个顺序就像所有列值都包含的一个唯一主键，逻辑上能够将被肢解出来的列值串在一起，知道它们是否属于同一条记录，这也是保证记录被拆分之后不会失真的一个重要手段。既然顺序是十分必要的、不能失真的因素，那么，当某条记录的某一列的值为空时就，我们就不能简单地跳过，必须显式地为其存储一个 NULL 值，以保证记录顺序的完整有效。但是，仅仅 NULL

值本身所能诠释的信息是不够的，比如，记录中某个 Name.Language.Country 列为空，由于 Name 和 Language 字段都是 repeated 类型，而 Country 又是 optional 类型，所以，这种情形可能表示 Country 字段没有值，也可能表示 Language 字段没有值，这两种情况在装配算法中是需要区分处理的，不能失真，所以才需要引出定义深度，能够准确描述出这种情形。例如，我们看到 r1 没有 Backward 链接，而 Links 字段是定义了的（在级别 1），为了保护此信息，我们就需要为 Links.Backward 列添加一个 NULL 值，并设置其定义深度为 1，说明 Links 字段是有定义的。类似地，在 r2 中值为 NULL 的 Name.Language.Country 字段的定义深度为 1，因为，虽然 Country 字段没有值，但是，Name 字段是有定义的，而 Name 字段的级别是 1，所以，Name.Language.Country 的定义深度为 1；同样，在 r1 中值为 NULL 的两个 Name.Language.Country 字段的定义深度分别为 2（在 Name.Language 内）和 1（在 Name 内）。

- 编码（Encoding）

每列存储为一组块。每个块包括重复深度和定义深度以及压缩的字段值。NULL 是由定义深度来决定的，所以它不会显式地存储。字段路径对应的定义深度小于路径上可选和可重复字段个数总和的，即可认为是 NULL。如果值是有被定义的，那么它的定义深度也不会被存储。类似地，重复深度只在必要时存储。比如，定义深度 0 意味着重复深度 0，所以后者可省略。事实上，图 13-3 中，没有为 DocId 存储深度。深度被打包为 bit 序列。我们只使用必需的位；比如，如果最大定义深度是 3，我们只需使用 2 个 bit。

13.3.2 将记录转换为列式存储

上面我们展示了使用列式格式表达出记录结构并进行编码。我们要面对的下一个挑战是如何高效地构造 column-stripe（图 13-1 中的右边部分），以及如何计算得到重复深度和定义深度。

计算重复深度和定义深度的基础算法在图 13-4 中给出。算法遍历记录结构然后计算每个列值的深度，当列值为缺失值时也不例外。在 Google 公司的各种应用中，经常会有一个 schema 包含了成千上万的字段，却只有其中少量的几百个字段在记录中被使用。因此，我们需要尽可能廉价地处理缺失字段。为了构造 column-stripe，我们创建一个树状结构，节点为 fieldWriter，它的结构与 schema 中的字段深度相符。基本的想法是，只在 fieldWriter 获得对应字段的值时才执行更新，而不尝试往下传递父节点的状态，除非绝对必要。子节点 fieldWriter 继承父节点的深度值。当任意值被添加时，一个子 fieldWriter 将深度值同步到父

节点。具体算法如图 13-4 所示。

```
1 procedure DissectRecord(RecordDecoder decoder,
2             FieldWriter writer, int repetitionLevel):
3   Add current repetitionLevel and definition level to writer
4   seenFields = {} // empty set of integers
5   while decoder has more field values
6     FieldWriter chWriter =
7       child of writer for field read by decoder
8     int chRepetitionLevel = repetitionLevel
9     if set seenFields contains field ID of chWriter
10      chRepetitionLevel = tree depth of chWriter
11    else
12      Add field ID of chWriter to seenFields
13    end if
14    if chWriter corresponds to an atomic field
15      Write value of current field read by decoder
16      using chWriter at chRepetitionLevel
17    else
18      DissectRecord(new RecordDecoder for nested record
19        read by decoder, chWriter, chRepetitionLevel)
20    end if
21  end while
22 end procedure
```

图 13-4 将一个记录分解为多个列的算法

图 13-4 展示的是算法如何将一个记录分解为多个列，也就是扫描记录后，计算得到每个字段值的重复深度和定义深度。子过程 DissectRecord 需要一个 RecordDecoder 参数，RecordDecoder 用于遍历二进制记录。FieldWriters 的深度结构和 schema 一致。对于每条记录来说，根 FiledWriter 将作为算法的参数，同时将 repetitionLevel 设为 0。DissectRecord 过程的主要工作就是维护当前的 repetitionLevel。当前的 definitionLevel 是由当前的 writer 在 schema 中所处的位置决定的，设为字段路径上可选字段和重复字段的个数的总和。

算法中的 While 循环（第 5 行）重复迭代所有原子的类型或者记录类型的字段。集合 seenFileds 跟踪记录中是否已经出现过某个字段，同时用来标识最近重复出现的那个字段。chRepetitionLevel 被设置为最近重复字段的 RepetititonLevel，默认值为父亲 RepetitionLevel 的值（9-13 行）。可以看到过程 DissectRecord 被重复地调用。

每个非叶子 writer 都维护着一系列深度（重复深度和定义深度）。同时每个 writer 都附带一个版本号。简单来说，当一个深度增加时，该 writer 的版本号就会递增 1。这样有利于子 writer 高效地记住父亲的版本号。如果一个子 writer 想要得到自己的值（非空），它只要和父亲 writer 同步，随后就能获得新的的深度信息。

因为输入的数据通常包含几十万条记录，几千个字段，所以，在内存中保存这些信息

显然是不合理的。一些深度信息可以暂存到磁盘中的文件中。对于一个空记录的无损编码来说，非原子字段（例如图 13-2 中的 Name.Language）可能需要它们自己的 column stripes，这些 column stripes 只包含深度，而没有非空的值。

13.3.3 记录的装配

从列式数据高效地装配记录，对于面向记录的数据处理工具（例如 MapReduce）而言是很重要的。给定一个字段的子集，我们的目标是重组原始记录就好像它们只包含选择的字段，其他字段就当不存在。核心想法是：我们为字段子集创建一个有限状态机（FSM），读取字段值和深度，然后顺序地将值添加到输出结果上。一个字段的 FSM 状态对应这个字段的 reader。状态的变化标记上了重复深度。一旦一个 reader 获取了一个值，我们将查看下一个值的重复深度来决定状态如何变化、跳转到哪个 reader。对于每一条记录，FSM 都是从开始状态到结束状态变化一次。

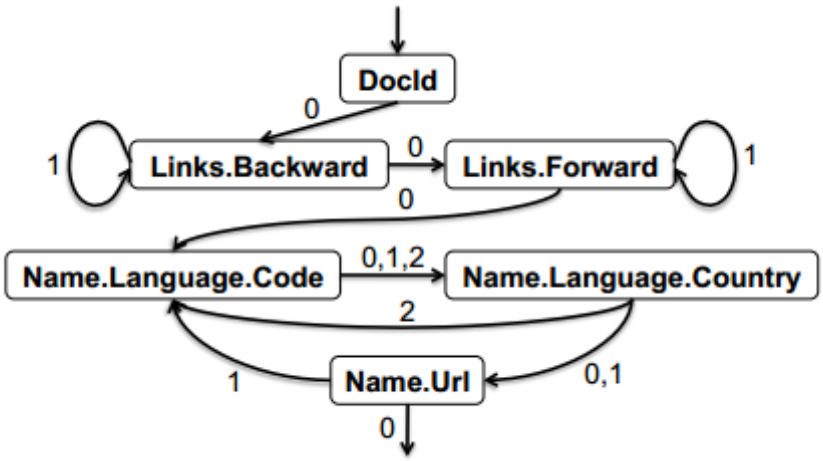

图 13-5 完整记录装配自动机

图 13-5 以 Document 为例展示了一个 FSM 重组一条完整记录的过程。开始状态是 DocId。一旦一个 DocId 值被读取，FSM 就转移到 Links.Backward。获取完所有重复字段 Backward 的值后，FSM 会跳向 Links.Forward，依此类推。这里要明确三个思路：第一，所有数据都是按图 13-3 那种类似一张张“表”的形式存储的；第二，算法会结合 schema，按照一定次序一张张地读取某些“表”（不是所有的，比如只统计 Forward 那就只会读取这一张“表”），次序是不固定的，这个次序也就是状态机内状态变迁的过程；第三，无论次序多么不固定，它都是按记录的顺序不断循环的（比如当前数据按顺序存储着 r1,r2,r3… ，那么，

就会进入第一个循环读取并装配出 r1，然后，进入第二个循环装配出 r2…），一个循环就是一个状态机从开始到结束的生命周期。通过对上面三点的思考，我们可以想到，在扫描过程中，需要不断做一件非常重要的事情——扫描到某张“表”的某一行时要判断这一行是不是属于下一条记录了，如果是，那么为了继续填充当前记录，就需要跳至下一张“表”继续扫描另一个字段值，否则就用此行的值装配当前记录，如此重复直到需要跳出最后一张“表”，一次循环结束（一个状态机结束，一条记录被装配完毕，进入下一个循环）。理解了这一点，就能理解为何要用状态机来实现算法了，因为循环内就是不断进行状态判断的过程。再深入思考一下，可以想到这个判断不仅是简单的“是否属于下一条记录”，对于 repeated 字段的子孙字段，还需要判断是否属于同一个记录的下一个祖先、并且是哪个层次的祖先。这里举一个例子，比如当前正在装配 r1 中的某个 Name 的某个 Language，扫描到了 Name.Language.Country 的某一行，如果此行重复深度为 0，表示属于下一条记录，说明当前 Name 下 Language 不会再重复了（当前 Name 的所有 Language 装配完毕），于是跳至 Name.Url 继续装配其他属性；如果为 1，表示属于 r1 的下一个 Name，也说明当前 Name 下 Language 不会再重复（当前 Name 的所有 Language 装配完毕），那也跳到 Name.Url；如果为 2，表示属于当前 Name 的下一个 Language（当前 Name 的 Language 还未装配完毕），那就走一个小循环，跳回上一个 Name.Language.Code 以装配当前 Name 的下一个 Language。

FSM 的构造逻辑可以这么表示：令 *l* 为当前字段读取器为字段 *f* 所返回的下一个重复深度。在 schema 树中，我们找到它在深度 *l* 的祖先，然后选择该祖先节点的第一个叶子字段 *n*。这样的 FSM 状态变化可以简写为(*f*;*l*)->*n*。比如，让 *l*=1 为 *f*=Name.Language.Country 读取的下一个重复深度。它的重复深度为 1 的祖先是 Name，它的第一个叶子字段是 *n*=Name.Url。

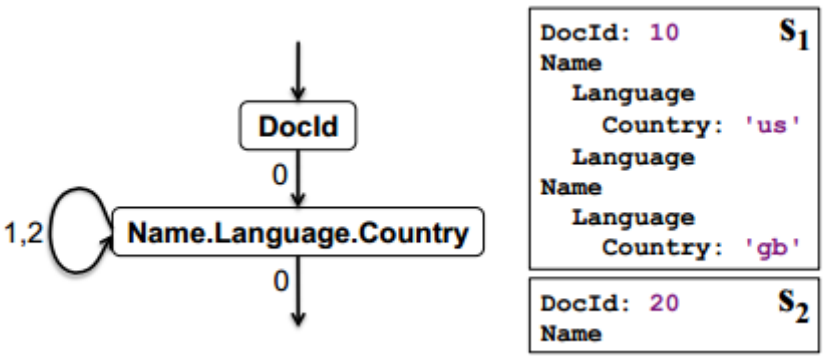

图 13-6 从两个字段中组装出记录的自动机及其组装结果

如果只有一个字段子集需要被处理，FSM 则更简单。图 13-6 描述了一个 FSM，读取

字段 DocId 和 Name.Language.Country。图中展示了输出记录 s1 和 s2。注意，Dremel 的编码和装配算法保护了字段 Country 的封闭结构。这个对于应用访问过程很重要，比如，Country 出现在第二个 Name 的第一个 Language，在 XPath 中，就可以用此表达式访问：/Name[2]/Language[1]/Country。

13.4查询语言

Dremel 的查询语言基于 SQL，可在列式嵌套存储上高效地执行，这里简介一下查询语言的特点。每个 SQL 语句（被翻译成代数运算）以一个或多个嵌套表格和它们的 schema 作为输入，输出一个嵌套表格和它的 schema。图 13-7 描述了一个查询例子，执行了投影、选择和记录内聚合等操作。例子中的查询执行在图 13-2 中的 t = {r1,r2}表格上。字段是通过路径表达式来引用。查询最终根据某种规则产出一个嵌套结构的数据，不需要用户在 SQL 中指明构造规则。

```
SELECT DocId AS Id,
  COUNT(Name.Language.Code) WITHIN Name AS Cnt,
  Name.Url + ',' + Name.Language.Code AS Str
FROM t
WHERE REGEXP(Name.Url, '^http') AND DocId < 20;
```

```
Id: 10                                  t1
Name
  Cnt: 2
  Language
    Str: 'http://A,en-us'
    Str: 'http://A,en'
Name
  Cnt: 0
```

```
message QueryResult {
  required int64 Id;
  repeated group Name {
    optional uint64 Cnt;
    repeated group Language {
      optional string Str; }}}
```

图 13-7 简单查询及其查询结果与输出的模式

为了解释这条查询究竟做了什么，我们考虑“选择”操作（WHERE 语句）。一个嵌套的记录可以看作一棵标记树，每个标记代表一个字段。选择操作会把不满足条件的树的分支裁剪掉。因此，只有那些 Name.Url 有定义的、且以‘http’开头的嵌套记录才会被取出来。接下来考虑“投影”操作，每个 SELECT 语句中的数值表达式会在与嵌套中重复最多输入字段的同层产生一个值。因此，字符串连接表达式在输入 schema 的 Name.Language.Code 层（即深度为 3 时）生成 Str 对应的值。Count 表达式说明了记录内的聚合操作。聚合操作在每个 Name 的子记录里完成，并产生一个 64 位的非负整形值表示 Name.Language.Code 在每个 Name 出现的次数。

此语言支持嵌套子查询、记录内聚合、top-k（排序）、joins（多表关联）和用户自定义函数等等。

13.5查询的执行

本节描述在数据分布式存储之后，如何尽可能并行地执行计算过程。核心概念就是实现一个树状的执行过程，将服务器分配为树中的逻辑节点，每个层次的节点履行不同的职责，最终完成整个查询。整个过程可以理解成一个任务分解和调度的过程。查询会被分解成多个子任务，子任务调度到某个节点上执行，该节点可以执行任务返回结果到上层的父节点，也可以继续拆解更小的任务调度到下层的子节点。此方案称为服务树（serving-tree）结构。

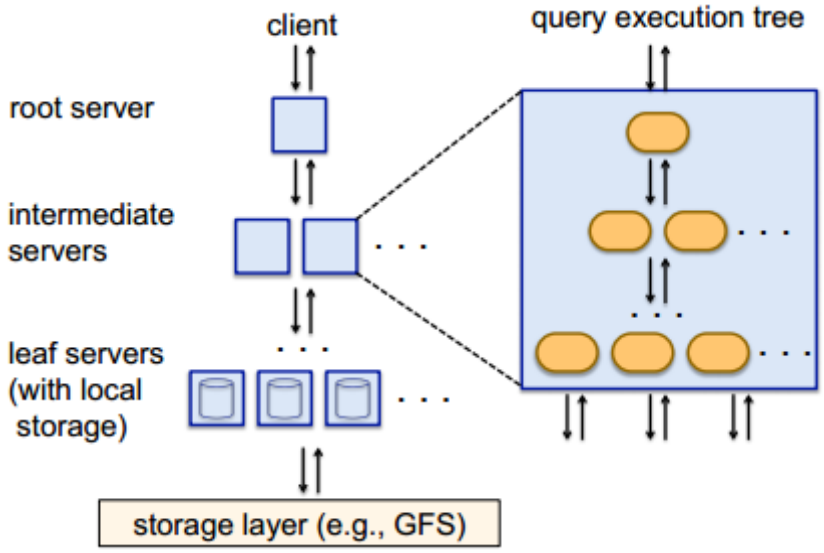

图 13-8 系统架构和在一个服务器节点内部的执行

- **树结构**

Dremel 使用一个多层次服务树来执行查询（见图 13-8）。一个根节点服务器接收到来的查询，从表中读取元数据，将查询路由到下一层。叶子服务器负责与存储层通讯，或者直接在本地磁盘访问数据。

一个简单的聚合查询如下：

SELECT A, COUNT(B) FROM T GROUP BY A

当根节点服务器收到上述查询时，它确定出所有 tablet，也就是表格的水平分割（把一个 column-stripe 理解成一个 table，称之为 T，table 被分布式存储和查询时可认为对 T 进行了水平拆分，tablet 就相当于 T 的一个分区），重写查询为如下：

SELECT A,SUM(c) FROM (R_1^1 UNION ALL ... R_n^1) GROUP BY A

R_1^1到R_n^1是树中第 1 层的（1 到 n）节点返回的子查询结果：

R_i^1=SELECT A, COUNT(B) AS c FROM T_i^1 GROUP BY A

T_i^1可认为是 T 在第 1 层的服务器 i 上被处理时的一个水平分区（tablet）。每一层的节点所做的都是与此相似的重写（rewrite）过程。查询任务被一级级地分解成更小的子任务，最终落实到叶子节点，并行地对 T 的 tablet 进行扫描。在向上返回结果的过程中，中间层的服务器担任了对子查询结果进行聚合的角色。此计算模型非常适用于返回较小结果的聚合查询，这种查询也是交互式应用中最常见的场景。大型的聚合或者其他类型的查询可能更适合使用并行 DBMS 和 MR 来解决。

- **查询分发器**

Dremel 是一个多用户系统，多个查询通常会被同时执行。一个查询分发器会基于查询任务的优先等级和负载均衡对查询任务进行调度。它还能帮助实现容错机制，当一个服务器变得很慢或者一个 tablet 备份不可访问时可以重新调度。

每个查询任务的数据处理量通常比可执行的处理单元（slot）的数量要多。一个 slot 对应一个叶子服务器上的一个执行线程。比如，一个 3000 个叶子服务器的系统，每个叶子服务器使用 8 个线程，则拥有 24000 个 slot。所以，一个 table 分解为 100000 个 tablet，则会分配大约 5 个 tablet 到每个 slot。在查询执行时，查询分发器会统计各 tablet 的处理耗时。如果一个 tablet 耗时较长或不成比例，它会被重新调度到另一个服务器。一些 tablet 可能需要被重新分发多次。

叶子服务器读取列式结构数据中的 stripe。每个 stripe 的块被异步预取；预读缓存通常命中率为 95%。tablet 一般复制三份。当一个叶子服务器读取其中一个备份失败时，它就会去读取另一个备份。

查询分发器有一个重要参数，它表示在返回结果之前一定要扫描百分之多少的 tablet，设置这个参数到较小的值（比如 98%而不是 100%）通常能显著地提升执行速度，特别是当使用较小的复制系数时。

本章小结

本章描述了一种能对大数据进行交互式分析的分布式系统——Dremel。Dremel 由几个简单是组件组成，是一种通用的、管理可扩展数据的解决方案，能在短时间内完成对大规模

数据的交互式查询与分析。同时，Dremel 具有很强的可扩展性、稳定性，它实现了对 MapReduce 的一种互补。

参考文献

[1]Melnik, Sergey, et al. "Dremel: interactive analysis of web-scale datasets." Proceedings of the VLDB Endowment 3.1-2 (2010): 330-339.

[2] 经典论文翻译导读之《Dremel: Interactive Analysis of WebScale Datasets》. http://blog.csdn.net/macyang/article/details/8566105

[3] 颜开. Google Dremel 原理－如何能 3 秒分析 1PB. http://blog.jobbole.com/29561/#jtss-tsina

附录 1:作者介绍

林子雨(1978－),男,博士,厦门大学计算机科学系助理教授,主要研究领域为数据库,数据仓库,数据挖掘.
主讲课程：《大数据技术基础》
办公地点：厦门大学海韵园科研 2 号楼
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu
数据库实验室网站：http://dblab.xmu.edu.cn